

Міністерство освіти і науки України  
Житомирський державний технологічний університет  
Факультет інформаційно-комп'ютерних технологій  
Кафедра інженерії програмного забезпечення

**ПОЯСНЮВАЛЬНА ЗАПИСКА**  
до магістерської атестаційної роботи  
на тему: «АНАЛІЗ ЕФЕКТИВНОСТІ JS-ФРЕЙМОРКІВ ДЛЯ  
WEB-РОЗРОБКИ»

Виконав студент 2-го курсу, групи ЗПІ-12-1м  
спеціальності 121 «Інженерія програмного  
забезпечення»

\_\_\_\_\_ О.Г. Гродецький  
Керівник к.ф-м.н., ст.викладач кафедри ІПЗ

\_\_\_\_\_ Л.В. Рудюк  
Рецензент к.т.н., доцент, кафедри ІПЗ

\_\_\_\_\_ О.І. Грабар

Житомир – 2018

ЖИТОМИРСЬКИЙ ДЕРЖАВНИЙ ТЕХНОЛОГІЧНИЙ  
УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

(назва прописними)

КАФЕДРА ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

(назва прописними)

СПЕЦІАЛЬНІСТЬ 121 «ІНЖЕНЕРІЯ ПРОГРАМНОГО  
ЗАБЕЗПЕЧЕННЯ»

(назва прописними)

ЗАТВЕРДЖУЮ

Зав. кафедри \_\_\_\_\_

\_\_\_\_\_  
(назва кафедри)

«01» жовтня 2018 р.

**ЗАВДАННЯ**

на магістерську атестаційну роботу

Студента Гродецького Олександра Геннадійовича

Тема роботи: Аналіз ефективності js-фреймворків для web-розробки

Затверджена Наказом університету від «27» жовтня 2018 р. № 462с

Термін здачі студентом закінченої роботи 06 грудня 2018 року

Вихідні дані роботи (зазначається предмет і об'єкт дослідження)

Об'єктом дослідження є результати аналізу для бізнесу і IT індустрії в цілому, так як додатки з високою продуктивністю дозволяють зберегти клієнтів і залучити нових.

Предметом дослідження є представники ресторанного бізнесу і IT індустрії в цілому.

Консультанти з атестаційної роботи магістра із зазначенням розділів, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Кравченко С.М.	01.09.2018	01.09.2018
2	Кравченко С.М.	15.10.2018	15.10.2018
3	Кравченко С.М.	05.11.2018	05.11.2018

Керівник \_\_\_\_\_  
(підпис)

### Календарний план

№ з/п	Назва етапів магістерської атестаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка задачі	01 жовтня 2018 року	виконано
2	Аналіз предметної області	12 жовтня 2018 року	виконано
3	Аналіз аналогів програмного продукту	20 жовтня 2018 року	виконано
4	Пошук та опрацювання літературних джерел	01 листопада 2018 року	виконано
5	Проектування структури	16 листопада 2018 року	виконано
6	Розробка методів для підняття ефективності	22 листопада 2018 року	виконано
7	Написання програмного коду	29 листопада 2018 року	виконано
8	Написання пояснювальної записки	05 грудня 2018 року	виконано
9	Захист	16 грудня 2018 року	

Студент

\_\_\_\_\_  
(підпис)

Керівник

\_\_\_\_\_  
(підпис)

## РЕФЕРАТ

Магістерська атестаційна робота містить: 60 сторінки, 16 ілюстрацій, 3 таблиці, список використаної літератури з 25 пунктів.

У даній роботі викладена сутність підходу до створення веб-сайту з високою продуктивністю на основі використання компонентного підходу. Дано загальні поняття про тестування продуктивності. Проведено аналіз продуктивності. Вивчено технологія тестування продуктивності за швидкістю завантаження сайту.

Робота буде корисна для front-end розробників, новачків які в майбутньому хочуть бути web-розробниками, так і для осіб, які цікавляться інтернет-технологіями.

Ключові слова: ВЕБ-СЕРВІС, ПРОДУКТИВНІСТЬ, ВЕБ-ДОДАТКИ, ТЕСТУВАННЯ, ФРЕЙМВОРК.

## ABSTRACT

Attestation master's work contains 60 pages, 15 figures, 2 tables, the list of used sources of 25 titles.

In this paper, the essence of the approach to creating a high performance website based on the use of the component approach is described. The general concepts of performance testing are given. Productivity analysis conducted. The technology of testing of productivity at the speed of site loading is studied.

The work will be useful for front-end developers, newcomers who in the future want to be web developers, and for those who are interested in Internet technologies.

Keywords: WEBSITE, PRODUCTIVITY, WEB APPLICATIONS, TESTING, FRAMEWORK.

## ЗМІСТ

<i>ВСТУП</i> .....	6
<i>РОЗДІЛ 1. АНАЛІЗ МЕТОДІВ ДЛЯ ЗБІЛЬШЕННЯ ПРОДУКТИВНОСТІ ВЕБ-ДОДАТКІВ</i> .....	8
1.1. Архітектура клієнт-серверної взаємодії .....	8
1.2. Порівняння аналогів .....	12
1.2.1. Інструменти для тестування продуктивності на стороні сервера ....	12
1.3. Прогресивне завантаження. ....	14
1.4. Оптимізація коду. ....	15
Висновки до розділу 1 .....	22
<i>РОЗДІЛ 2. РОЗРОБКА АЛГОРИТМІВ АНАЛІЗУ ПРОДУКТИВНОСТІ ВЕБ-ДОДАТКІВ</i> .....	23
2.1. Показники продуктивності веб-додатків .....	23
2.1. Вартість JavaScript .....	27
Висновки до розділу 2. ....	35
<i>РОЗДІЛ 3. РЕАЛІЗАЦІЯ ВЕБ-СЕРВІСУ</i> .....	37
3.1. Розробка веб-додатку.....	37
3.2. Проведення вимірів.....	42
3.3. Результати проведення тестів продуктивності та їх аналіз .....	47
Висновки до розділу 3 .....	55
<i>ВИСНОВКИ</i> .....	57
<i>ЛІТЕРАТУРНІ ДЖЕРЕЛА</i> .....	59

## ВСТУП

**Актуальність теми.** З розвитком світових технологій експоненціально збільшується вибір девайсів (комп'ютерів, ноутбуків, планшетів, смартфонів), зі зростаючою продуктивністю, тому необхідно створювати продуктивні додатки з невеликими вкладеннями.

Сьогодні Інтернет - це web-ресурси мережі, доступні за http - гіпертекстовий протоколу. Сьогодні web-мережу вступила в своє третє покоління і настільки швидко розвивається, що кожні 10 років ці покоління змінюються.

Сьогодні завдяки розвитку оптоволоконних комунікаційних технологій стала можливою масова передача відеоінформації, сьогоднішні комерція і фінанси немислимі без web.

Web індустрія займає велику частину всесвітнього ринку. Це багатомільярдні компанії, що займаються розробкою web-додатків та програмного забезпечення, що складаються з мільйонів рядків програмного коду і файлів мультимедіа.

Front-end і back-end - терміни в програмній інженерії, які розрізняють відповідно до принципу поділу відповідальності між представницьким рівнем і рівнем доступу до даних відповідно. Front-end - інтерфейс взаємодії між користувачем і основний програмно-апаратною частиною (back-end). Front-end і back-end можуть бути розподілені між однією або декількома системами.

В архітектурі програмного забезпечення може бути багато рівнів між апаратною частиною і кінцевим користувачем, кожен з яких також може мати front-end і back-end. Front end - це абстракція, яка надає користувацький інтерфейс.

**Метою даної роботи** є тестування і аналіз продуктивності на прикладі веб-додатку для ресторану, а також виявлення найбільш оптимальних підходів реалізації сайтів з високою продуктивністю.

**Для вирішення даної проблеми** були поставлені наступні завдання:

- Розробити веб-додаток для ресторану.

- Провести порівняльний аналіз продуктивності на основі швидкості завантаження сайту.
- Дослідити ефективність застосування запропонованих технологій, методів і підходів на основі аналізу їх використання.

**Наукова новизна** полягає в наступних розробках і дослідженнях автора:

- Архітектура веб-додатку, яка за рахунок застосування компонентної моделі, що дозволяє повторне використання коду, і за рахунок ізоляції різних модулів системи знизити трудомісткість розробки і ресурсомісткість масштабних веб-додатків.
- Розробка методів збільшення продуктивності будь-яких веб-додатків.

**Об'єктом дослідження** є результати аналізу для бізнесу і ІТ індустрії в цілому, так як додатки з високою продуктивністю дозволяють зберегти клієнтів і залучити нових.

**Предметом дослідження** є представники ресторанного бізнесу і ІТ індустрії в цілому.

**Публікація.** За темою атестаційної магістерської роботи опубліковано тези:

1. О.Г. Гродецький, магістр, гр. ЗПІ-12-1м, Ю.О. Кубрак. Мова есмаcript. Тези І Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення» (19–20 жовтня 2018 р.). – Житомир : Вид. О.О. Євенок, 2018. – С.13.

## РОЗДІЛ 1. АНАЛІЗ МЕТОДІВ ДЛЯ ЗБІЛЬШЕННЯ ПРОДУКТИВНОСТІ ВЕБ-ДОДАТКІВ

### 1.1. Архітектура клієнт-серверної взаємодії

Веб-додатки представляють собою тип програм, побудованих за архітектурою “клієнт-сервер”. Клієнт-серверна модель являє собою структуру програми, яка розподіляє завдання і навантаження між постачальниками ресурсів і послуг, званим серверами і тими, хто користуються цими послугами, званими клієнтами. По суті клієнти і сервери являє собою програмне забезпечення. Як правило вони розташовані на різних обчислювальних машинах, і обмінюються даними по обчислювальній мережі за допомогою мережових протоколів, але іноді клієнт і сервер можуть перебувати на одному комп'ютері. Хост сервера запускає одну або кілька серверних програм, які розподіляють свої ресурси між клієнтами. Клієнт запитує вміст сервера, але не передає нічого. Сервери очікують запити, а клієнти ініціюють сеанси зв'язку з ними.

Запити клієнта обробляються на сервері - там, де розташована База Даних і Система Управління Базою Даних (СУБД). Це дає перевагу у відсутності пересилання великих обсягів даних, а також запит оптимізується таким чином, що на нього витрачається мінімум часу. Все це підвищує швидкодію системи і знижує час очікування результату запиту. При виконанні запитів сервером суттєво підвищується ступінь безпеки даних, оскільки правила цілісності даних визначаються в базі даних на сервері і є єдиними для всіх додатків, що використовують цю БД.

Функції клієнта:

- ініціалізація запиту серверу
- обробка результатів запитів, отриманих від сервера
- представлення результатів запиту користувачеві у формі інтерфейсу користувача

Функції сервера:



- прийом запитів від клієнта
- обробка запитів
- виконання запитів до Бази Даних і їх оптимізація
- відправка результатів запитів клієнта
- забезпечення системи безпеки
- забезпечення стабільності на багато користувачів режиму роботи

### Продуктивність

Розробка програмного забезпечення, розвивалася роками, починаючи з ручного тестування, але в ті часи вимоги були набагато нижче, сайти були текстові і завантажувалися протягом декількох хвилин. Тому у веб- розробників було набагато менше стимулів для попереднього тестування. Але ставки виросли, як тільки електронна комерція набрала обертів в світі веб-розробки. Тому тестування стали проводити в середовищі розробки. Але з ростом додатків почали автоматизувати і тестування.

Розробники почали писати автоматизовані тести.

Зрештою, тестування дозріло до такої міри, що воно поширилося за межі простих наборів тестових модулів і інтеграційних тестів в стилі відтворення. Організації почали будувати все більш витончені, тонкі тестові комплекти.

На сьогоднішній день, ставки для роботи додатків стали вище, ніж будь-коли. Тести роботи програми вже давно стали стандартом де-факто. Так як додатки стали занадто складними для ручного тестування, були створили тестові фреймворки, за допомогою яких можна було автоматизувати тестування. І будь-який хороший код починається з написання тестів. Але це не дозволяє дізнатися, як буде вести себе додаток в дикому середовищі. Тестування продуктивності веб- додатків виправляє це.

Тестування продуктивності - це форма тестування програмного забезпечення, в якому основна увага приділяється тому, як система працює під певним навантаженням. Тестування продуктивності повинно дати організаціям діагностичну інформацію, необхідну їм для виявлення і усунення вузьких місць.

Повільна робота додатків впливає на те, що платні користувачі йдуть, а нових передплатників стає менше, що впливає на доходи.

Найчастіше вирішувати проблеми з продуктивністю виявляється дуже важко в зв'язку з тим, що розробникам складно відтворювати такого роду “баги”. Проблеми з продуктивністю безпосередньо не зачіпають поведінку програмного забезпечення. Швидше, вони пов'язані з тим, як програмне забезпечення реагує на хаотичний світ середовищ, в яких запускається додаток. Тому необхідно проводити тестування продуктивності.

Звичайне QA тестування полягає в спостереженні, як додаток поводить себе з однією людиною. Для виробництва тестування необхідна симуляція суворих умов, таким чином можна виявити, як веде себе додаток під великим навантаженням, так зване тестування навантаження.

У тестовій середовищі можна вибрати навантаження для додатка, наприклад, одночасне використання додатка тисячею користувачами при виконанні звичайних операцій і вимір поведінки програми. Зберігає він чуйність або сповільнюється або навіть падає? Звичайно, таке тестування не проводитиметься тисячею реальних людей. Для цього створюєте програмне забезпечення, яке допомагає імітувати навантаження.

На додаток до тестування навантаження проводиться стрес-тестування і тестування на витривалість. При стрес-тестуванні з додатком створюються несприятливі умови, щоб побачити, як він себе веде в цих ситуаціях. Це дозволяє зрозуміти, в який момент воно зламається і які вузькі місця у додатка є. Кожна програма має точку переривання, тому після відмови на стрес-тесті можна вносити зміни, знаючи що шукати і коли очікувати проблем, і зробити все можливе, щоб в той момент, коли додаток терпів невдачу, він робив це витончено і розумно.

При тестуванні на витривалість застосовується навантаження на певний час - бажано на довго. Так само, як необхідно знати, як веде себе додаток з великою кількістю користувачів, необхідно знати, як він буде вести себе

протягом декількох тижнів або місяців. І краще це дізнатися в тестовому середовищі, ніж в режимі реального часу.

Тестування продуктивності краще проводити в процесі розробки програми, так як є ймовірність виявити збої, викликані архітектурними рішеннями.

Перед початком тестування продуктивності необхідно зрозуміти виробничі умови роботи програми. Необхідно з'ясувати нормальні і пікові умови для підготовки до різних видів тестування продуктивності. Тестування на продуктивність буде включати тестування навантаження, стрес тестування, тестування на витривалість та інші. При тестуванні навантаження вибирається умови виробництва (наприклад, кількість користувачів, обсяг трафіку та інше). Потім імітується ця навантаження, щоб побачити, як система обробляє її. Також можна збільшити навантаження до точки злому, щоб побачити, в який момент це відбувається, і як цю проблему можна вирішити.

Після визначення умови тестування програми, перше завдання полягатиме в налаштуванні середовища для тестування.

Необхідно симулювати виробництво тестування не тільки на машині розробника в міру своїх можливостей. Це може означати більш жорсткі сервери, ніж ті, що стоять в компанії.

Після настройки тестового оточення необхідно з'ясувати, як імітувати умови виробництва для навантажувальних і стрес-тестів. При проведенні тести продуктивності, необхідний запис результатів, пропуски і збої для створення статистики.

Традиційна перевірка при звичайному тестуванні досить проста в деякому сенсі. “Коли ми вводимо X, програма повинна повернути Y.”

При тестуванні продуктивності все набагато складніше. Так як немає справжніх і несправжніх результатів, а скоріше є межі нормальної роботи програми та рекомендації. Наприклад, можна сказати, що існує будь-яке припустиме - максимальне час відповіді додатку. Найчастіше при початку проведення тестування продуктивності можна і не знати, що очікувати, але в ході

роботи необхідно протестувати продуктивність програми як на клієнті (в браузері), так і на сервері.

З цієї причини необхідно встановити вихідні дані. Перевіряючи, як додаток працює як є, збирати дані, задавши якісь базові передбачувані дані і виправляти додаток до тих пір, поки воно не буде їм відповідати. І далі запускати тести, змінюючи вихідні дані, створюючи більш несприятливі умови, виправляти додаток і так далі.

## 1.2. Порівняння аналогів

### 1.2.1. Інструменти для тестування продуктивності на стороні сервера

Apache Bench and Siege відмінно підходять для тестів швидкої навантаження з одного кінцевого пункту. Якщо необхідно просто отримати уявлення про запити в секунду для кінцевої точки, це відмінні рішення.

Locust.io - платформа тестування навантаження на стороні сервера. з відкритим вихідним кодом, яка дозволяє виконувати складні транзакції і може з легкістю генерувати високий рівень паралелізму.

Bees with Machine Guns - автори описують, як “утиліту” для створення великої кількості примірників мікро EC2 для навантажувального тестування веб-додатків.

Multi-Mechanize - це середовище з відкритим вихідним кодом для тестування продуктивності і навантаження, яка запускає паралельні сценарії Python для генерації навантаження (синтетичних транзакцій) щодо віддаленого сайту або служби. Вона зазвичай використовується для тестування продуктивності мережі і масштабованості, але її також можна використовувати для створення робочого навантаження для будь-якого віддаленого API, доступного з Python.

Siege - ця утиліта тестування і тестування продуктивності HTTP була розроблена, щоб дозволити веб-розробникам вимірювати код під навантаженням, щоб побачити, як він буде себе вести при навантаженні в

Інтернеті. Siege підтримує базову автентифікацію, файли cookie, протоколи HTTP і HTTPS і дозволяє користувачеві набирати веб-сервер з налаштованим кількістю імітованих веб-браузерів.

Apache Bench - інструмент для тестування HTTP-сервера Apache, щоб отримати уявлення про те, як працює Apache.

Httpperf - цей інструмент вимірює продуктивність веб-сервера і забезпечує гнучку генерацію різноманітних робочих навантажень HTTP і продуктивності сервера.

JMeter - для перевірки продуктивності як на статичних, так і на динамічних ресурсах (файли, сервлети, скрипти Perl, об'єкти Java, бази даних і запити, FTP-сервери і т.д.). Також його можна використовувати для симуляції великого навантаження на сервер, мережа або об'єкт, щоб перевірити додаток міцність або проаналізувати загальну продуктивність при різних типах навантаження.

Інструменти для тестування продуктивності на стороні клієнта

Сучасні програми витрачають більше часу на стороні клієнта, ніж на сервері.

Google PageSpeed Insights - служба, яка аналізує вміст веб-сторінки і генерує рекомендації, для більш швидкого завантаження сторінок. Скорочення часу завантаження сторінки знижує частоту відмов і збільшує коефіцієнт конверсії.

Sitespeed.io - інструмент для оцінки продуктивності на клієнтській стороні з реальних браузерів. Цей інструмент з відкритим вихідним кодом аналізує швидкість і продуктивність веб-сайту на основі кращих практик і показників часу.

Розробники не завжди можуть змінити додатки для оптимізації продуктивності на стороні клієнта. Google інвестував у створення ngx\_pagespeed і mod\_pagespeed як розширення веб-серверів для автоматизації підвищення продуктивності без необхідності зміни коду.

Google ngx\_pagespeed серверний модуль nginx з відкритим вихідним кодом і Google mod\_pagespeed сервер HTTP з відкритим вихідним кодом Apache

прискорюють роботу сайту і скорочують час завантаження сторінки, застосовує кращі методи веб-продуктивності для сторінок і пов'язаних з ними файлів (CSS, JavaScript, зображень), що не вимагаючи зміни існуючого контенту або робочого процесу.

WebPagetest.org - забезпечує глибоке розуміння продуктивності клієнтської сторони в різних реальних браузерах. Ця утиліта перевіряє веб-сторінку в будь-якому браузері, з будь-якого місця, з будь-якого мережевого умові.

Ґрунтуючись на проведених дослідженнях, можна виявити такі методики для збільшення продуктивності веб-додатків, які необхідно дотримуватися при розробці.

### 1.3. Прогресивне завантаження.

Багато сайтів оптимізують видимість контенту як дорогу інтерактивність. Щоб отримати швидко першу інтерактивність, коли у проекту великі пакети JavaScript, розробники іноді використовують рендеринг на стороні сервера; потім “оновлюють” його, щоб приєднати обробники подій, коли JavaScript, нарешті, буде завантажений.

Але слід бути обережним - у цього підходу є свої мінуси:

- зазвичай відправляється більш великий відповідь з HTML, який може відкласти інтерактивність,
- може залишити користувача в проміжному стані, де половина сайту може не бути інтерактивною до тих пір, поки весь JavaScript закінчить обробку.

Прогресивна завантаження може бути кращим підходом. Для цього відправляється мінімально функціональна сторінка (що складається тільки з HTML / JS / CSS, необхідного для поточного шляху). У міру надходження більшої кількості ресурсів додаток може поступово дозавантажувати і розблокувати додаткові функції.

Підводячи підсумки можна сказати, що розмір переданого коду має вирішальне значення для визначення часу завантаження. Час парсинга залежить від процесора пристроїв.

Дуже важливо передбачати на стадії розробки архітектурних рішень для JS, для якого використання буде додаток, якщо створюється сайт, орієнтований на мобільні пристрої, необхідно зробити все можливе, щоб розробка велася на обладнанні, на якому час парсинга / компіляції JavaScript досить велика, в такому випадку буде простіше приймати ефективні рішення для економії витрат на завантаження JavaScript.

#### 1.4. Оптимізація коду.

Для поліпшення швидкості на стороні сервера зазвичай використовуються наступні оптимізації:

Мови програмування, такі як PHP, Perl, Python або ASP, як правило, об'єднуються з такими базами даних, як MySQL, PostgreSQL або Microsoft SQL Server, для створення таких програм, як WordPress, Drupal, Magento і всілякі настраюються платформи.

Це програмне забезпечення, як правило, досить оптимізовано з коробки, але часто буває багато налаштувань коду або плагінів, які показують низьку продуктивність в результаті неефективного коду або не оптимізованих запитів до бази даних.

Оптимізація коду включає в себе аналіз запитів коду і бази даних і пошук місць, де код неефективний і де запити до бази даних повільні. Знайшовши ці “гарячі точки”, завдання розробника усунути ці проблеми. Для коду це часто передбачає пошук кращого алгоритму чи внесення змін до коду для роботи у вузьких місцях. Для баз даних це може включати додавання індексів для прискорення запиту, переписування запиту або зміна структури бази даних.

Також на стороні клієнта при складанні бандла можна використовувати такі плагіни, як uglifier для скорочення коду, і необхідно відстежувати і видаляти невикористаний код.

## **Оптимізація черговості стилів і скриптів.**

Правильне впорядкування зовнішніх таблиць стилів і зовнішніх і вбудованих скриптів дозволяє краще розпаралелити завантаження і прискорити час рендеринга браузера.

Оскільки код JavaScript може змінювати вміст і розташування веб-сторінки, браузер затримує рендеринг будь-якого контенту, наступного за тегом сценарію, до тих пір, поки цей скрипт не завантажиться, проаналізовано і виконаний. Проте, що більш важливо для тимчасового відключення, багато браузери блокують завантаження ресурсів, згаданих в документі після сценаріїв, до тих пір, поки ці сценарії не будуть завантажені і виконані. З іншого боку, якщо інші файли вже завантажуються при зверненні до файлу JS, файл JS завантажуватиметься паралельно з ними.

### **Відкладене завантаження JavaScript (defer).**

Щоб завантажити сторінку, браузер повинен проаналізувати вміст всіх тегів `<script>`, що додає додатковий час на завантаження сторінки. Мінімізуючи кількість JavaScript, необхідне для рендеринга сторінки, і скасовуючи парсинг непотрібного JavaScript до тих пір, поки він не буде виконаний, можна зменшити проміжок часу завантаження своєї сторінки.

Існує кілька методів, які можна використовувати для відстрочки синтаксичного аналізу JavaScript. Найпростіший і кращий метод - просто відкласти завантаження JavaScript `<script defer>` до тих пір, поки він не знадобиться. Другий спосіб - використовувати атрибут `<script async>`, де це необхідно, що запобігає розбір парсинга від початкового завантаження сторінки, відкладаючи його до тих пір, поки потік користувальницького інтерфейсу браузера не буде зайнятий чимось іншим. Якщо жоден з цих методів не підходить, існують деякі додаткові методи, зазвичай використовуються в мобільних додатках, які описані нижче.

При створенні мобільних додатків може знадобитися завантажити весь JavaScript, необхідний для додатка, це необхідно для того, щоб додаток могло продовжувати працювати, коли користувач знаходиться в автономному режимі.



У цьому випадку деякі програми, такі як мобільний Gmail, вважають корисним завантажувати JavaScript в коментарях, а потім викликати `eval ()` JavaScript, коли це необхідно. Такий підхід гарантує, що весь JavaScript завантажиться під час початкового завантаження сторінки, не вимагаючи того, щоб JavaScript був розпарсений.

Альтернативою зберігання коду в коментарях є зберігання коду в строкових літералах JavaScript. При використанні цього методу JavaScript обробляється тільки при необхідності, знову шляхом виклику `eval ()` в строковому літералі. Цей метод також дозволяє додатку завантажувати JavaScript раніше, але відкласти синтаксичний аналіз до тих пір, поки він не знадобиться.

Необхідно враховувати, що переміщення тегів `<script>` в кінець сторінки є не оптимальним, так як браузер буде продовжувати показувати індикатор зайнятості, поки сторінка не завершить розбір цього JavaScript. Користувачі можуть дочекатися, поки індикатор завантаження сторінки не покаже, що завантаження сторінки завершена, перш ніж взаємодіяти зі сторінкою, тому важливо завантажувати JavaScript таким чином, щоб мінімізувати час, необхідний браузеру для вказівки того, що завантаження сторінки завершена.

Існує статистика, що на сучасних мобільних пристроях кожен додатковий кілобайт JavaScript додає близько 1 мс часу розбору до загального часу завантаження сторінки. Таким чином, 100 КБ JavaScript, включений в завантаження початкової сторінки, додасть 100 мс часу завантаження для користувачів. Оскільки JavaScript повинен аналізуватися при кожному відвідуванні сторінки, це додатковий час завантаження буде частиною завантаження кожної сторінки, завантаженої з мережі, через кеш браузера або в автономному режимі HTML5.

### **Оптимізація завантажуються зображень.**

Для зменшення часу завантаження сторінки, найкраще використовувати зображення з відповідним розміром, це особливо ефективно для малопотужних (наприклад, мобільних) пристроїв.

Як правило у розробників може виникнути бажання використовувати зображення з високою роздільною здатністю для всіх типів і розмірів дисплеїв з різною щільністю пікселів. Стиснення зображень часто дає менші розміри файлів з кращою якістю, тому не варто забувати про стиснення зображень, однак необхідно мати на увазі, що зображення з більш високою роздільною здатністю завжди вимагають більше пам'яті і вимагають більше часу для декодування, тому важливо також враховувати малопотужні пристрої зі стандартними дисплеями. З цієї причини рекомендується використовувати атрибут `srcset`. Підтримка цього атрибута браузером дуже хороша.

Іноді може знадобитися відображати одне й те саме зображення в різних розмірах, тому розробники використовують один ресурс зображення і за допомогою HTML або CSS на сторінці, масштабують зображення.

Наприклад, у користувача може бути 10-кратна міні-версія великого зображення розміром 250 x 250 пікселів, і замість того, щоб змусити користувача завантажувати два окремі файли, розробники використовують розмітку для зміни розміру мініатюрної версії. Це має сенс, якщо фактичний розмір зображення відповідає хоча б одній - найбільшому примірнику на сторінці, в даному випадку 250 x 250 пікселів. Однак, якщо використовувати зображення, яке більше розмірів, що використовуються в усіх варіантах дозволу екранів, користувачеві відправляються непотрібні байти. Тому дуже важливо використовувати редактор зображень для масштабування зображень відповідно до найбільшим розміром, необхідним для сторінки веб-додатку.

Але цей підхід досить стандартний, і більш для більш витонченого зменшення часу завантаження сторінки, можна реалізувати ледачу завантаження зображень - для цього необхідно написати компонент який дозволить нашій додатком працювати наступним чином - спочатку на сторінку будуть завантажуватися зображення, зменшені до розміру 4 рх по меншій з сторін, після того, як відбудеться подія `loaded` - розрахунковий час (тобто сторінка буде завантажена), за допомогою `javascript` почнуть завантажуватися повні версії зображень, як тільки повна версія і розуміння завантажиться, в тезі `img` шлях в

атрибуті src на зменшену версію зображення підмінити на шлях до зображень в хорошій якості. Тим самим для завантаження сторінки будуть використані дуже маленькі і легкі зображення, що істотно скоротить час завантаження

#### Кешування сторінок.

Більшість веб-сайтів на сьогоднішній день є динамічними, це означає, що вони витягують з бази даних інформацію, і додають дані в шаблони сторінок. Це відбувається кожного разу, коли користувач запитує сторінку з сервера, і час, необхідний для виконання цього процесу, залежить від ефективності коду і потужності зазначених серверів.

Оскільки сервер обробляє тисячі запитів для однієї і тієї ж сторінки і по суті “створює” одну і ту ж сторінку кожен раз, чому б не створити сторінку один раз і відправляти цю “попередньо побудовану” версію кожен раз, коли її хтось запитує? Це називається кешуванням сторінок.

Кешування сторінок може бути дуже ефективним засобом прискорення генерації сторінок, але також має і свої недоліки:

Сторінки, що вимагають автентифікації, не можуть бути кешованими (Оскільки вони містять інформацію про користувачів).

Зміни на сторінках не відображаються до закінчення терміну дії кешу сторінок.

Також для зменшення завантаження сторінок необхідно збільшити час простій браузерного кеша.

Час завантаження сторінки може бути значно покращено, попросивши відвідувачів зберегти і повторно використовувати файли, включені в ваш сайт. Це особливо ефективно на сайтах, де користувачі регулярно переглядають ті ж області веб-сайту Це називається кешування браузера.

Кожен раз, коли браузер завантажує веб-сторінку, він повинен завантажувати всі веб-файли, щоб правильно відображати сторінку. Сюди входять всі HTML, CSS, JavaScript і зображення.

Деякі сторінки можуть складатися тільки з декількох файлів і бути маленькими за розміром - може бути, пара кілобайт. У інших може бути багато файлів, і вони можуть становити до декількох мегабайт.

Ці великі файли займають більше часу для завантаження і можуть бути особливо відчутні, якщо користувачі знаходяться на повільному інтернет-з'єднанні (або мобільному пристрої).

Кожен файл робить окремий запит на сервер. Чим більше запитів сервер отримує одночасно, тим більше роботи він повинен виконувати, що впливає на подальше зниження швидкості завантаження сторінки.

Кешування браузера може допомогти, зберігши деякі з цих файлів локально в браузері користувача. Під час першого візиту на сайт будуть завантажені файли, коли користувач повертається на сайт, оновлює сторінку або навіть переміщається на іншу сторінку сайту, у нього вже є деякі файли, які йому потрібні локально.

Це означає, що кількість даних, які браузер користувача завантажує, менше, а значить і на сервер потрібна менша кількість запитів. В результаті зменшено час завантаження сторінки.

Кешування браузера працює шляхом маркування певних сторінок або частин сторінок, оскільки їх необхідно оновлювати з різними інтервалами. Наприклад, логотип на веб-сайті практично ніколи не змінюється. Кешірую цей логотип, можна повідомити браузеру користувача завантажувати це зображення раз в тиждень. При кожному відвідуванні протягом тижня, не вимагатиметься завантаження логотипу.

Веб-сервер, що пропонує браузеру зберігати ці файли, а не завантажувати їх, економить час користувачів і пропускну здатність веб-сервера.

Основна причина, по якій кешування браузера дуже важливо, полягає в тому, що кешування знижує навантаження на веб-сервер, що в кінцевому підсумку знижує час завантаження для користувачів.

Щоб включити кешування браузера, необхідно відредагувати HTTP-заголовки, щоб встановити час закінчення терміну дії для певних типів файлів.

Також необхідно встановити Apache для обслуговування відповідних заголовків.

Для цього буде потрібно файл .htaccess в корені домену. Цей файл є прихованим файлом, але він повинен відображатися на FTP-клієнтів, таких як FileZilla або CORE. Необхідно відредагувати це файл .htaccess.

.htaccess (з точкою на початку імені) - це файл-конфігуратор Apache-серверів, який дає можливість конфігурувати роботу сервера в окремих директоріях (папках), котрі дають доступу до головного конфігураційного файлу (apache / conf / httpd.conf). Наприклад, встановлювати права доступу до файлів в директорії, змінювати назви індексних файлів, самостійно обробляти помилки Apache, перенаправляючи відвідувачів на спеціальні сторінки помилок. .htaccess є звичайним текстовим документ, розширення якого htaccess.

У цьому файлі встановлюються параметри кешування, щоб повідомити браузеру, які типи файлів будуть кешуватися.

```
## EXPIRES CACHING ##
<IfModule mod_expires.c>
ExpiresActive On
ExpiresByType image / jpg “доступ плюс 1 рік”
ExpiresByType image / jpeg “доступ плюс 1 рік”
ExpiresByType image / gif “доступ плюс 1 рік”
ExpiresByType image / png “доступ плюс 1 рік”
ExpiresByType text / css “доступ плюс 1 місяць”
ExpiresByType application / pdf “доступ плюс 1 місяць”
ExpiresByType text / x-javascript “доступ плюс 1 місяць”
ExpiresByType application / x-shockwave-flash “доступ плюс 1 місяць”
ExpiresByType image / x-icon “доступ плюс 1 рік”
ExpiresDefault “доступ плюс 2 дня”
</ IfModule>
## EXPIRES CACHING ##
```

Залежно від файлів веб-сайту можна встановити час закінчення терміну дії (“протухання”). Якщо деякі типи файлів оновлюються частіше, необхідно встановити для них більш короткий час “протухання” (наприклад для файлів css)

Для всіх статичних ресурсів термін дії необхідно встановлювати не менше одного місяця (рекомендується: доступ плюс 1 рік)

Але важливо також не встановлювати кешування більш ніж на рік вперед.

Так як при зміні файлів на веб-сайті, у користувачів можуть все ще відображатися старі файли, і вони не зможуть отримати нову версію сайту після оновлень.

### Висновки до розділу 1

В першому розділі було проаналізовано методи для збільшення продуктивності веб-додатків, а саме:

1. Прогресивне завантаження;
2. Оптимізація коду;
3. Оптимізація черговості стилів і скриптів, відкладене завантаження JavaScript.

Було здійснено порівняння аналогів аналітичних систем клієнт-серверної архітектури.

## РОЗДІЛ 2. РОЗРОБКА АЛГОРИТМІВ АНАЛІЗУ ПРОДУКТИВНОСТІ ВЕБ-ДОДАТКІВ

### 2.1. Показники продуктивності веб-додатків

Дослідження продуктивності веб-додатку проводилось за наступними показниками:

- час на редирект
- час відповіді сервера
- час до першого байта
- завантаження DOM (вмісту сторінки - контенту)
- час до інтерактиву (коли користувач може почати взаємодіяти зі сторінкою)
- час повного завантаження

Коли виконується запит для сторінки, Front-end і Server-side компоненти витрачають певну кількість часу для виконання своїх операцій. Оскільки ці операції по суті послідовні, їх сумарний час можна вважати загальним часом завантаження сторінки.

Нижче наведено короткий огляд ключових показників ефективності:

- Менш 100 мс сприймається як миттєвий відповідь;
- Від 100 мс до 300 мс - спостерігається незначна затримка;
- Одна секунда - це межа для того, щоб потік думки користувача залишався безперервним;
- Користувачі очікують, що сайт завантажиться через 2 секунди;
- Через 3 секунди 40% відвідувачів покинуть ваш сайт;
- 10 секунд - це обмеження для підтримки уваги користувача.

На рисунку 1 зображено час завантаження сторінки.



Рис. 2.1. Час завантаження сторінки.

### I. Час на редирект

Час на редирект - цей час, витрачений на переспрямування URL- адрес до завантаження останньої HTML-сторінки.

Загальні переадресації включають:

- перенаправлення URL-адреси, що починається ні з `www`, на `www` (наприклад, `example.com` на `www.example.com`),
- перенаправлення на захищений URL (наприклад, `http: //` в `https: //`),
- перенаправлення для установки файлів `cookie`,
- перенаправлення на мобільну версію сайту.

### II. Тривалість з'єднання

Деякі сайти можуть також виконувати ланцюжок з декількох переадресацій (наприклад, спочатку на `www`, а потім на безпечний URL- адресу). Час на редирект - це загальна кількість часу, витрачений на переспрямування, воно дорівнює 0, якщо не було переадресації.

Переадресація складається з часу з початку твору вимірювання до моменту, коли ми почнемо запит останньої HTML-сторінки (коли ми отримаємо перший відповідь 200 OK).

До цього часу екран браузера порожній. Тому дуже важливо стежити за скороченням часу на редирект сторінки, зводячи до мінімуму переадресації.

Як тільки все переадресації завершені, вимірюється тривалість з'єднання. Це час, витрачений на підключення до сервера, для виконання запиту на сторінку.

З технічної точки зору, ця тривалість являє собою комбінацію заблокованого часу, часу DNS, часу з'єднання і часу відправлення запиту (а не просто часу з'єднання).



Тривалість підключення складається з усього цього, включаючи час відправки в остаточному запиті HTML-сторінки (перша відповідь 200 OK).

Весь цей час екран браузера також залишається порожнім. Цьому можуть сприяти різні причини, в тому числі повільний зв'язок між тестовим сервером і сайтом або повільне час відгуку з сайту.

### III. Час роботи сервера

Як тільки з'єднання буде завершено і буде виконаний запит, серверу необхідно сформулювати відповідь для сторінки. Час, що витрачається на генерацію відповіді, називається часом роботи сервера.

Час роботи сервера складається з часу очікування в запиті сторінки.

Існує ряд причин, за якими тривалість роботи сервера може бути досить повільним. Навіть після того, як оптимізована продуктивність на стороні Front-end, необхідна оптимізація і серверної частини. Це означає оптимізацію генерації сторінки сервером (час до першого байта). Як правило, цей час має бути в межах однієї секунди (або якомога менше).

Є багато причин повільної роботи серверної сторони, але вони по суті можуть бути згруповані у дві категорії:

- неефективний код або SQL
- вузькі місця / повільний сервер

Оскільки кожен сайт має унікальну платформу і настройку, рішення цих проблем для кожного сайту своя. Можливо, одного сайту необхідно оптимізувати серверний код, а іншому може знадобитися більш потужний сервер. Також важливі бюджетні обмеження, таким чином оптимізація коду на стороні сервера для незначного збільшення швидкості може бути більш доступною, ніж оновлення серверів для збільшення швидкості.

### IV. Час до першого байта (TTFB)

Час до першого байта (TTFB) - це загальна кількість часу, що витрачається на отримання першого байта відповіді після виконання запиту. Це сума “Тривалість редиректу” + “Тривалість з'єднання” + “Тривалість роботи сервера”. Цей показник є одним з ключових показників продуктивності мережі.

Деякі способи поліпшення TTFB включають в себе: оптимізацію коду програми, реалізацію кешування, точну настройку конфігурації веб-сервера або оновлення серверного обладнання.

#### V. Час першої відтворення

Час першої відтворення - це перший момент, коли браузер робить будь-якої рендеринг на сторінці. Залежно від структури сторінки ця може бути просто колір фону (включаючи білий), або це може бути велика частина відображається сторінки. Це час має важливе значення, оскільки до цього моменту браузер буде показувати тільки порожню сторінку, і це зміна дає користувачеві вказівку, що сторінка завантажується. Однак ми не знаємо, на скількох сторінках сталася перше відтворення, тому на всіх сторінках додатку відтворення не обов'язкове.

#### VI. Завантаження DOM

Завантаження DOM - це момент, коли браузер завершив завантаження і аналіз HTML, і була створена DOM (Document Object Model). DOM - це те, як браузер структурує HTML, щоб він міг його відображати. Завантаження DOM дуже близька до часу завантаження вмісту DOM.

#### VII. Час до першого інтерактиву

Або час завантаження вмісту DOM (DOM завантажено або DOM готове для стислості) - це точка, в якій DOM готовий (т. Е. DOM interactive), і немає таблиць стилів, які блокують виконання JavaScript.

Якщо немає таблиць стилів, які блокують виконання JavaScript, і немає синтаксичного аналізатора, блокуючого JavaScript, то це буде те ж саме, що і інтерактивне час DOM.

Багато фреймворків JavaScript використовують цю подію в якості відправної точки для початку виконання свого коду.

Оскільки ця подія часто використовується JavaScript в якості відправної точки, а затримки в цю подію означають затримки в відтворенні, важливо переконатися, що порядок стилів і сценаріїв оптимізований і що розбір JavaScript відкладений.

#### VIII. Час повного завантаження

Час завантаження - це коли обробка сторінки завершена, і всі ресурси на сторінці (зображення, CSS і т. Д.) Завершили завантаження. Це також той же самий час, коли DOM готовий і відбувається подія JavaScript `window.onload`.

Необхідно звернути увагу, що також може бути JavaScript, який ініціює подальші запити для отримання більшої кількості ресурсів, отже для результатів для нас краще час повного завантаження.

## 2.1. Вартість JavaScript

Оскільки зараз пишуться сайти, які в більшій мірі залежать від JavaScript, доводиться платити за дані, які відправляються на сайт, але це не завжди можна легко помітити. Тому дуже важливо, щоб сайт завантажувався і швидко працював на мобільних пристроях. Коли більшість розробників думають про вартість JavaScript, вони думають про це з точки зору вартості завантаження і виконання. Відправлення більшої кількості байтів JavaScript по кабелю займає тим більше часу, ніж повільніше з'єднання користувача.

Розглянемо такий приклад сторінки:

```
<html>
<head>
  <link rel="stylesheet" href="/styles.css">
  <script src="/app.js" async></script>
</head>
<body>
<my-app>
  <picture slot="hero-image">
    <source srcset="img@desktop.png, img@desktop-2x.png 2x" media="(min-width:
    990px)">
    <source srcset="img@tablet.png, img@tablet-2x.png 2x" media="(min-width:
    750px)">
    <img srcset="img@mobile.png, img@mobile-2x.png 2x" alt="I don't know why.
    It's a perfectly cromulent word!">
  </picture>
</my-app>
</body>
</html>
```

### Листінг 2.1.

Браузер отримує цей документ від сервера у відповідь на GET запит.

Сервер відправляє його як потік байтів, і коли браузер стикається з кожним з підресурсів, на який посилається документ, він запитує їх.

Щоб ця сторінка була завантажена, вона повинна стати інтерактивною, тобто потрібно - “Час до інтерактиву”. Браузери обробляють введення користувача, генеруючи події DOM, які прослуховує код додатку. Ця обробка введення відбувається в основному потоці документа, де виконується JavaScript.

Існує ряд операцій, які можуть проходити в інших потоках, дозволяючи браузеру залишатися чуйним:

- Аналіз HTML
- Розбір CSS
- Аналіз і компіляція JavaScript (іноді)
- Деякі завдання збірки сміття JS
- Декодування зображень
- Перетворення і анімації CSS
- Прокрутка основного документа

Наступні ж операції повинні виконуватися в основному потоці:

- виконання JavaScript
- побудова DOM
- створення макета
- обробка введення

Виконання скриптів затримує інтерактивність декількома способами:

Якщо сценарій виконується більше 50 мс, то час до інтерактиву затримується на весь час, необхідний для завантаження, компіляції та виконання JS.

Будь-DOM або призначений для користувача інтерфейс, створений в JS, недоступний для використання до запуску скрипта.

З іншого боку, зображення не блокують основний потік, не блокують взаємодію при аналізі або разтруванні і не перешкоджають тому, щоб інші частини призначеного для користувача інтерфейсу залишалися інтерактивними.

Тому, дуже важливо розуміти, що 150 КБ JS буде затримувати час до інтерактиву через:

- Запитувати код, включаючи DNS, TCP, HTTP і накладні витрати на декомпресію
- Парсинг і компіляцію функцій верхнього рівня JS виконання скрипта

Виходячи з проведеної статистики 45% мобільних підключень доводиться на 2G по всьому світу, а 75% з'єднань відбувається на 2G або 3G.

Медіанний користувач знаходиться в повільній мережі.

У Google Chrome DevTools при включенні тротлінга на повільному з'єднанні 3G - імітується зв'язок з пропускнуною спроможністю 400 мс RTT це швидкість 400-600 кбіт / с - таким чином можна прийняти ці показники за базові.

У сучасному світі розробки існує якийсь метричний показник, в межах якого повинно знаходитися час до інтерактиву:

- TTI менше 5 секунд для першого навантаження;
- TTI менше 2 секунд для подальших навантажень.

Виходячи з цього, можна зробити наступний розрахунок:

Пошук DNS і авторизації TLS становить близько 1,6 секунди, а це значить, що для завантаження сторінки залишається близько 3,4 секунди

Потім можна обчислити, скільки даних можна відправити за цим посиланням за 3.4 секунди:  $400 \text{ Кбіт} / \text{с} = 50 \text{ КБ} / \text{с}$ . Значить  $50 \text{ КБ} / \text{с} * 3,4 = 170 \text{ КБ}$ .

Підводячи підсумок, можна сказати, що в середньому бюджет для ресурсів, переданих по мережі (CSS, JS, HTML і дані) повинен складати 170 КБ, якщо ж розробники використовують js-framework, то бюджет знижується до 130 КБ, так як інфраструктура як правило займає близько 40 КБ.

Знизити вартість передачі JavaScript по мережі за допомогою таких дій (рис. 2.2):

- Відправлення тільки того коду, який потрібен користувачу. У даній ситуації може бути дуже корисно поділ коду.
- Мінімізація (Uglify для ES5, babel-minify або uglify-es для ES2015)



Рис. 2.2. Рекомендації щодо скорочення кількості JavaScript.

- Сильне стиснення (з використанням Brotli ~ q11, Zopfli або gzip). Brotli перевершує gzip за ступенем стиснення. Компанії CertSimple вдалося скоротити на 17% JS файли, а LinkedIn заощадити 4% на часі завантаження.
- Видалення невикористаного коду. Визначити невикористаний код можна за допомогою DevTools code coverage. Для видалення невикористаного коду можна використовувати tree-shaking, Closure Compiler's і бібліотеки, такі як lodash-babel-plugin або ContextReplacementPlugin Webpack для бібліотек, таких як Moment.js. Також рекомендується використання babel-preset-env & browserlist в сучасних браузерях. Також необхідно проводити аналіз Webpack bundle для виявлення можливостей видалити невикористовувані залежності.
- Кешування JS файлів, для мінімізації кількості мережових запитів. Також необхідно визначити оптимальні терміни життя для скриптів (max-age) і токенів перевірки поставки (ETag), щоб уникнути передачі незмінених байтів. Кешування за допомогою Service Worker'ов може зробити мережу додатків стійкою і надасть швидкий доступ до таких функцій, як кеш-код V8.

Парсинг / Компіляція.

Після завантаження одне з найбільших - це час, коли JS-двигжок парсить / компілює JavaScript код.

Summary		Bottom-Up	Call Tree	Event Log
Filter		No Grouping ▼		
Self Time		Total Time		Activity
2.6 ms	27.8 %	2.6 ms	27.8 %	Major GC
1.4 ms	15.6 %	1.4 ms	15.6 %	DOM GC
1.4 ms	15.2 %	1.4 ms	15.2 %	Update Layer Tree
1.3 ms	14.6 %	1.3 ms	14.6 %	Hit Test
1.1 ms	12.5 %	1.1 ms	12.5 %	Composite Layers
0.8 ms	8.2 %	0.8 ms	8.2 %	DOM GC
0.2 ms	2.5 %	0.2 ms	2.5 %	Event
0.2 ms	2.1 %	0.2 ms	2.1 %	Recalculate Style
0.1 ms	1.4 %	0.1 ms	1.4 %	DOM GC

Рис. 2.3. Час на парсинг / компіляцію панелі Performance вкладки The Bottom- Up / Call Tree для головної сторінки сайту

У Chrome DevTools аналіз і компіляція - жовте часу “Scripting” на панелі Performance.

В панелі Chrome DevTools Performance далі Bottom-Up. З включеною статистикою Call Runtime V8 (рисунок 2.3) ми можемо побачити час, витрачений на такі фази, як парсинг і компіляція.

На рисунку 4 наведено сумарне співвідношення витраченого часу на розбір отриманих даних.



Рис. 2.4. Співвідношення часу на завантаження і парсинг / компіляцію сторінки.

Великий час парсинга / компіляції коду можуть сильно впливати на те, як скоро користувач зможе взаємодіяти з сайтом. Чим більше JavaScript коду відправляється, тим більше часу необхідно для його парсинга і компіляція, тобто до моменту, коли сайт стане інтерактивним. У порівнянні з JavaScript також безліч витрат на обробку зображень з еквівалентним розміром (так як їх ще потрібно декодувати), але в середньому потрібно більше часу для обробки javascript коду, ніж для завантаження зображень.

Дуже важливо розуміти, що байти JavaScript і байти зображення мають дуже різні витрати. Зазвичай зображення не блокують основний потік або не перешкоджають взаємодії з інтерфейсом під час декодування і растрівання. Однак JS може затримувати інтерактивність через тимчасових витрат на парсинг, компіляцію і виконання.

Довгий парсинг і компіляції дуже важливі, коли мова йде про середні мобільних телефонах. В середньому у користувачів можуть бути телефони з повільними процесорами і графічними процесорами, без кеша L2 / L3, які також можуть мати обмежену пам'ять.

“Можливості мережі і можливості пристрою не завжди збігаються. Користувач, у якого дуже добре сполучення не обов'язково має кращий процесор для парсинга і компіляції JavaScript, відправленого на його пристрій. І навпаки, погане з'єднання з мережею, але швидкий процесор.” - Крістофер Бакстер, LinkedIn

На рисунку 5 відзначена вартість розбору ~ 1 МБ декомпресировать (простого) JavaScript на низькому і високому рівні апаратного забезпечення. Можна відзначити різницю в 2-5 разів для парсинга / компіляції коду між найшвидшими телефонами на ринку і середніми телефонами.

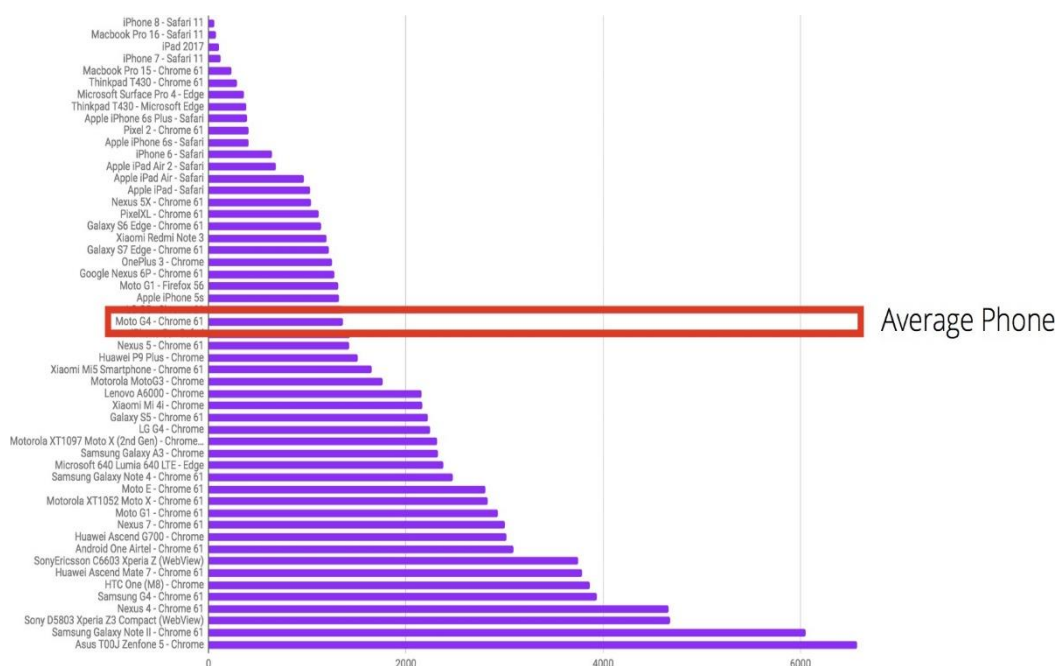


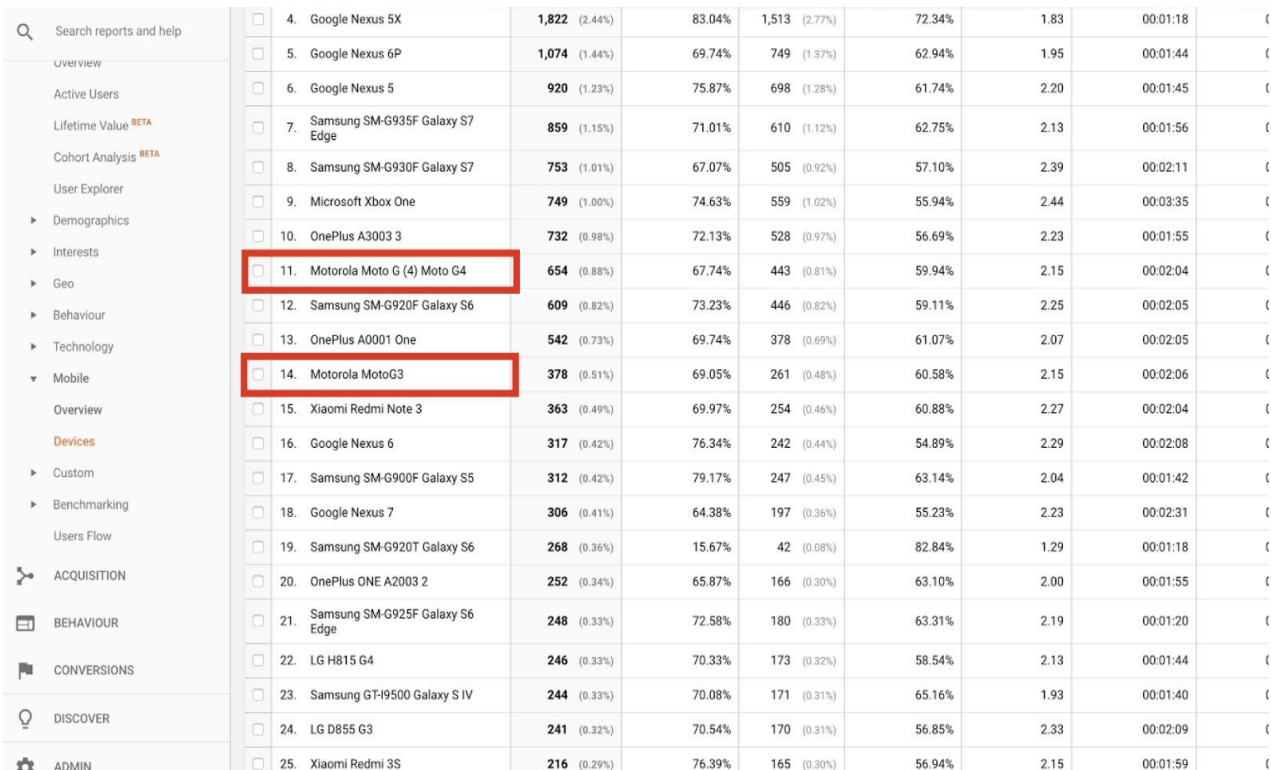
Рис. 2.5. Час парсинга 1 МБ bundle JavaScript (~ 250 КБ gzipped) на десктопний і мобільних пристроях різних класів за станом на 2018 рік.



Розглядаючи вартість парсинга на прикладі декомпресії розпакованих значень, наприклад, ~ 250 КБ gzipped JS, необхідно мати на увазі, що в розпакованому вигляді розмір може бути до 1 Мбайт коду.

Правильніше за все проводити тестування на середньому обладнанні, так як час парсинга і компіляції коду може бути в рази більше на смартфоні з середньою продуктивністю, ніж на останній версії iPhone. Аналітика може дати уявлення про класи мобільних пристроїв, за допомогою яких користувачі звертаються до сайту. Це може надати можливості для розуміння реальних обмежень ЦП / ГПУ, з якими вони працюють.

У HTTP-архіві (понад 500 тис. Сайтів) для аналізу завантаження JavaScript на мобільних пристроях, можна побачити, що близько 50% сайтів необхідно більше 14 секунд до інтерактивності (рисунок 2.6). Ці сайти витрачають до 4 секунд лише на парсинг і компіляцію JavaScript. Тому необхідно прагнути скоротити час завантаження сайтів.



Rank	Device	User Count	Percentage	Time to Interactivity
4.	Google Nexus 5X	1,822	(2.44%)	83.04%
5.	Google Nexus 6P	1,074	(1.44%)	69.74%
6.	Google Nexus 5	920	(1.23%)	75.87%
7.	Samsung SM-G935F Galaxy S7 Edge	859	(1.15%)	71.01%
8.	Samsung SM-G930F Galaxy S7	753	(1.01%)	67.07%
9.	Microsoft Xbox One	749	(1.00%)	74.63%
10.	OnePlus A3003 3	732	(0.98%)	72.13%
11.	Motorola Moto G (4) Moto G4	654	(0.88%)	67.74%
12.	Samsung SM-G920F Galaxy S6	609	(0.82%)	73.23%
13.	OnePlus A0001 One	542	(0.73%)	69.74%
14.	Motorola MotoG3	378	(0.51%)	69.05%
15.	Xiaomi Redmi Note 3	363	(0.49%)	69.97%
16.	Google Nexus 6	317	(0.42%)	76.34%
17.	Samsung SM-G900F Galaxy S5	312	(0.42%)	79.17%
18.	Google Nexus 7	306	(0.41%)	64.38%
19.	Samsung SM-G920T Galaxy S6	268	(0.36%)	15.67%
20.	OnePlus ONE A2003 2	252	(0.34%)	65.87%
21.	Samsung SM-G925F Galaxy S6 Edge	248	(0.33%)	72.58%
22.	LG H815 G4	246	(0.33%)	70.33%
23.	Samsung GT-I9500 Galaxy S IV	244	(0.33%)	70.08%
24.	LG D855 G3	241	(0.32%)	70.54%
25.	Xiaomi Redmi 3S	216	(0.29%)	76.39%

Рис. 2.6. Статистика часу до інтерактиву на мобільних пристроях на основі HTTP-архіву

Видалення некритичного JavaScript може скоротити час на передачу коду, парсинг і компіляцію, а також потенційні витрати пам'яті.

Час виконання.

Це не просто парсинг і компіляція коду, які впливають на вартість. Виконання JavaScript (запуск коду парсинг / компіляції) є однією з операцій, яка виконується в основному потоці. Тривалий час виконання також впливає на те, як скоро користувач може взаємодіяти з сайтом.

Щоб вирішити цю проблему, необхідно розбивати JavaScript код на невеликі фрагменти, щоб уникнути блокування основного потоку.

Для того, щоб зберегти час парсинга / компіляції і часу передачі JavaScript коду, існують патерни, які можуть допомогти у вирішенні цих завдань, наприклад, route-based chunking або PRPL.

PRPL - це патерн, який оптимізує інтерактивність за допомогою агресивного кодового поділу і кешування.

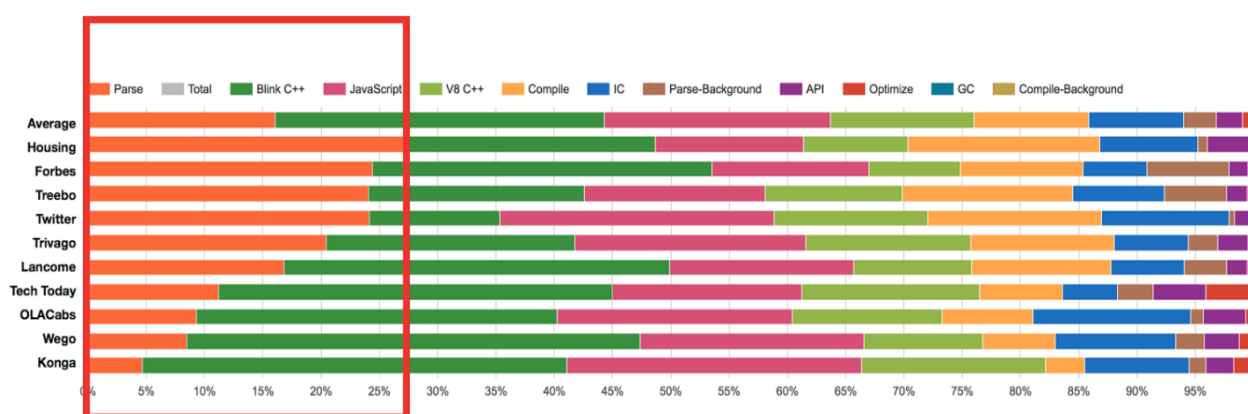


Рис. 2.7. Час завантаження популярних мобільних сайтів

На рисунку 2.7 наведеному вище відображено час завантаження популярних мобільних сайтів і прогресивних веб-додатків за допомогою V8 Runtime Call Stats. Як можна помітити, час парсинга (показано помаранчевим кольором) займає значну частину часу, яке витрачають багато з сайтів: Wego (<https://wego.com/>), сайту, який використовує PRPL, вдається підтримувати низький час на парсинг для своїх маршрутів, швидко отримуючи інтерактивність. Багато з перерахованих вище сайтів.

Інші витрати.

JavaScript може впливати на продуктивність сторінки іншими способами:

– Витік пам'яті. Сторінки можуть часто призупиняти свою роботу, тобто втрачати свою інтерактивність через збирача сміття (garbage collector), що дуже помітно в швидко реагують додатках. Основним недоліком збирачів сміття є недетермінованість, тобто складно зрозуміти в який момент може статися прибирання сміття. Коли браузер відновлює пам'ять, виконання JS призупиняється, тому браузер, часто збирає сміття, може призупиняти виконання частіше, ніж може бути зручно користувачам. Необхідно уникати витоків пам'яті і частих пауз збирачів сміття, щоб сторінки не втрачали свою інтерактивність. Існує чотири найпоширеніші види витоків пам'яті JavaScript: випадкові глобальні змінні, забуті коллбеки і таймери, посилання на віддалені з DOM елементи, замикання.

– Під час запуску, JavaScript з довгим часом виконання коду може блокувати основний потік, роблячи недоступними сторінки. Розбивка коду на більш дрібні частини (з використанням `requestAnimationFrame()` або `requestIdleCallback()` для планування) може мінімізувати проблеми з відгуком сайту.

## Висновки до розділу 2.

В другому розділі була проведена розробка алгоритмів аналізу продуктивності, а саме:

### 1. Дослідження продуктивності веб-додатку:

- час на редирект
- час відповіді сервера
- час до першого байта
- завантаження DOM
- час до інтерактиву
- час повного завантаження

### 2. JavaScript

Виконання скриптів затримує інтерактивність декількома способами:

Якщо сценарій виконується більше 50 мс, то час до інтерактиву затримується на весь час, необхідний для завантаження, компіляції та виконання JS.

Будь-DOM або призначений для користувача інтерфейс, створений в JS, недоступний для використання до запуску скрипта.

Було проаналізовано більшість методів для підвищення продуктивності веб-додатку, аналіз сервера, розділення коду, стиснення, мініфікація коду, завантаження картинок під використовуваний девайс та паттерни для того, щоб зберегти час парсинга коду.

## РОЗДІЛ 3. РЕАЛІЗАЦІЯ ВЕБ-SERVICES

### 3.1. Розробка веб-додатку на Gatsby.js

Перед розробкою проекту була створена Mind Map (рисунок 3.1).

Розробка веб-додатку почалася зі створення репозитарію на Github'і.

Код розроблявся на мові JavaScript та фреймворку GatsbyJS в редакторі Visual Studio Code.

Gatsby - це неймовірно швидкий фреймворк для розробки веб-сайтів на React. Він дозволяє створювати сайти, засновані на React буквально за лічені хвилини. Gatsby підходить для проектів різних масштабів - від блогів до корпоративних веб-сайтів.

Так як проекти, створені за допомогою Gatsby, засновані на React, їх сторінки, при взаємодії з ними користувача, що не перезавантажуються, що робить такі проекти дуже швидкими.

Для розробки була обрана бібліотека React, через використання цієї бібліотекою віртуального DOM'a, а також із-за реактивної і композитної компонентою структури, яка дозволяє з легкістю перевикористати елементи коду.

Як менеджер пакетів був обраний yarn. Для ініціалізації проекту в терміналі пишеться команда:

- yarn init - після чого в корені проекту створюється файл package.json, в якому вказується назва проекту, опису, версія, автор, ліцензія, прописуються залежності (встановлені за допомогою yarn пакети), а також скрипти, що запускають додаток.

Також для розробки був обраний Gatsby - генератор статичних сайтів для React - це набір файлів, які без серверної підтримки можуть працювати, як Single Page Application(SPA). Gatsby за допомогою запитів до API, написаних на GraphQL для кожної сторінки може витягувати дані, які потім використовуються для створення статичного клієнтського React- додатку.

Для встановлення Gatsby в терміналі необхідно написати наступну команду:

```
npm i -g gatsby-cli.
```

Для створення нового сайту: *gatsby new cuba-site*

Gatsby запускає середу розробки, доступну на localhost:8000 по команді в терміналі:

```
gatsby develop
```

*gatsby build* - Gatsby виконає оптимізовану збірку для сайту, генеруючи статичні HTML і зібрані JavaScript файли, іншими словами білд.

*gatsby serve* - Gatsby стартує локальний HTML сервер для тестування сайту.

Також в файлі package.json написані такі скрипти:

“Deploy”: “*gatsby build --prefix-paths && gh-pages -d public*” - для деплою проекту.

“Format”: “*prettier --trailing-comma es5 --no-semi --single-quote --write*” - для форматування коду (приведення до однаковості): звисають коми, відсутність точок-ком в кінці рядка і одиничні лапки.

Структура проекту зображена на рисунку 3.2.

Крім файлу package.json, в корені проекту створюється файл gatsby-config.js, в якому вказані Такий модуль gatsby-plugin-react-helmet - для роботи з React'ом, а також плагін gatsby-plugin-offline - для роботи в офлайн режимі.

Для управління і зберігання контенту був обраний сервіс contentful, для роботи з яким вам потрібен додаток для gatsby - gatsby-source-contentful, який також прописаний у файлі gatsby-config.js разом з spaceId і accessToken.

У contentful були створені моделі контенту і доданий контент для сторінок з наступними полями:

- Events / Сторінка Події (date, title, description, type, image);
- Gallery / Сторінка Галереї (title, type, image);
- Menu / Сторінка меню (title, description, price, image);
- Home Page Slider (зображення для слайдера на головній сторінці).

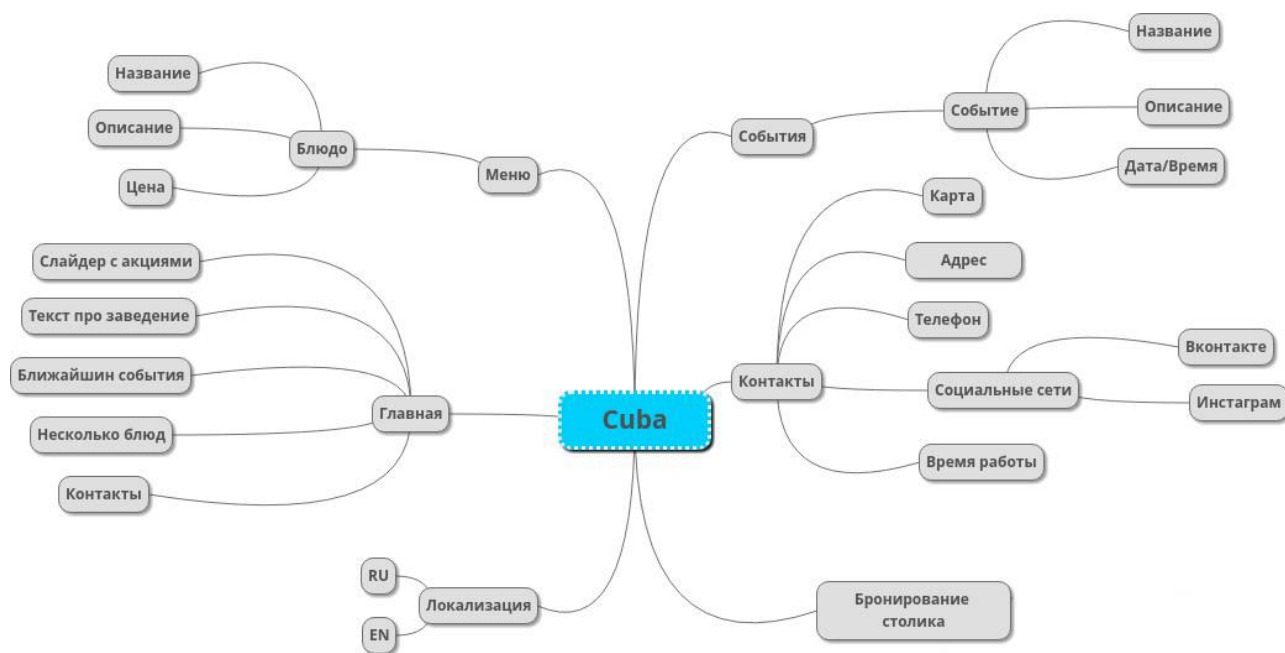


Рис. 3.1. Карта сайта (Mind Map).

Також для підтримки чистоти коду були обрані linter'и - `eslint` - `import` - для перевірки коду `es2015` +, а також для перевірки орфографічних помилок. Крім цього для розробки був доданий плагін `jsx-a11y` - для перевірки доступності елементів JSX.

У файлі `.stylelintrc` - вказані правила стилів.

Також додані два файли `.eslintignore` і `.stylelintignore` - для виключення файлів з перевірки, що знаходяться в директоріях `node_modules`, `assets`, `public` і `node_modules`, `assets`, `public`, `static`, `docs` відповідно.

Також для роботи з git'ом був доданий файл `.gitignore`, для вказівки файлів, які не повинні потрапити в коміт.

Для крос налаштувань редактора був доданий файл `.editorconfig`.

У файлі `browserslist` - зазначено, підтримка яких браузерів здійснюється.

В директорії `assets` зберігаються `svg` іконки.

Також в директорії `Modals` створені компоненти модального вікна для бронювання столиків - вибір дати, часу, кількості осіб, із зазначенням імені та номера телефону потенційного відвідувача.

Для написання коду були встановлені наступні залежності, які вказані в файлі `package.json`: `classnames` (для можливості вказівки декількох класів в `jsx` компонентах), `gatsby`, `gatsby-link`, `gatsby-plugin-offline`, `gatsby-plugin-react-helmet`, `gatsby-source-contentful` (для роботи з `gatsby`), `left-pad` (для додавання нулів на початку числа), `react-google-maps`, `react-input-range`, `react-responsive-carousel` (`react` - компоненти), `babel-eslint`, `eslint`, `eslint-plugin-compat`, `eslint-plugin-import`, `eslint-plugin-jsx-a11y`, `eslint-plugin-react`, `gh-pages`, `husky`, `lint-staged`, `prettier`, `prettier-eslint`, `stylelint`, `stylelint-config-standard` (Лінтера).

У файлі `pages-request.js` прописані запити в `contentful` на `graphql`.

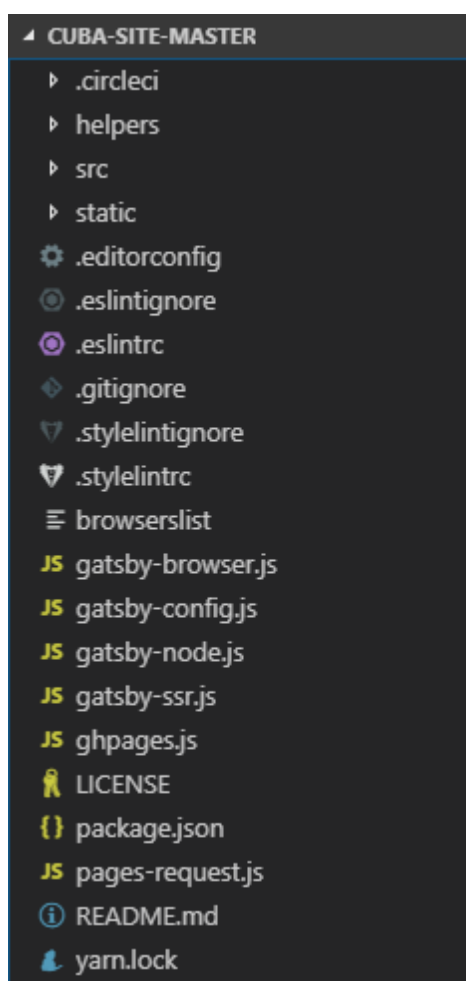


Рис. 3.2. Структура проекту

Файл `gatsby-ssr.js` - генерація статички (серверний рендер) - сервер отримує готовий `html`, тим самим відбувається швидше перший інтерактив.



Файл `gatsby-node.js` - для створення сторінок `events`, `menu`, `gallery` і одної події або одного пункту меню на основі відповіді після запиту, написаного на `graphql`

У файлі `gatsby-browser.js` - клієнтський рендер (`spr`)

У файлі `html.js` - створюється `html` документ для генерації статички, а також завантажується `google` аналітика.

Для зберігання стану програми був підключений `redux`. Він необхідний для перемикання локалізації. Для цього створені каталоги `state`, `actions`, `reducers`:

В директорії `store` `state.js` - `createStore` - створення `store`;

і `connectComponent.js` - підключення компоненти до `store`.

В директорії `reducer` - файл `locale.js`:

- `initial state` - початковий стан 2 ключа `locale` і `locales`, для кожного ключа повинен бути `reducer`
- `locale` - поточна локалізація, `locales` - масив можливих локалізацій
- `actions` - дії які викликають `reducer`.
- `helpers` - `combineNodes` - приходять дані з різними локалізаціями - для сортування на `ru` / `en`.

Основна директорія - `src`. У ній були створені такі директорії:

`layouts` - обгортка для всіх сторінок;

`pages` - директорія зі сторінками додатка - головна (`index.js`), контакти (`contacts.js`), події (`events.js`), галерея (`gallery.js`), меню (`menu.js`), сторінка не знайдена (`404.js`) . Також створено спільний файл стилів `style.module.css`

`modules` - директорія з модулями (функціями, які можуть бути використані в різних компонентах): `locales.js` (для перекладу сайту на англійську мову була написана функція `transformLocales`, завдяки якій, в кожній компоненті, де потрібен переклад тексту вказується переклад російською і англійською мовами) та модуль `date.js` (для роботи з датами і часом написані дві функції `getDate` і `getTime` відповідно)

`components` - директорія з компонентами. Був розроблений загальний компонент `Page`, від якого успадковуються всі сторінки і компонент `PageLayout`,

в якому розміщені загальні компоненти Footer і Header. А також розроблені наступні компоненти:

- Actions (в яких вказані номер телефону, адреса, час роботи і посилання на модальне вікно для бронювання столиків);
- CarouselWidget - віджет з зображеннями на головній сторінці;
- Navbar - панель навігації в header'е;
- Locale - перемикання мови;
- SVGIcon і Iconed - для svg іконок;
- PhotoCollage - колаж фотографій меню і подій на головній сторінці, що складається з компонент Poster;
- Picture - компонента завантаження і оптимізації зображень;
- Social - компонента svg іконок і посилань на соціальні мережі;
- GoogleMap - компонента карти зі сторінки контакти;
- PageGrid - сітка для розміщення зображень з відповідним контентом (компоненти EventTile, MenuTile, GalleryTile) на сторінках подій, меню і галереї;
- SinglePost - компонента шаблон

Сайт написаний з використанням адаптивної верстки, тим самим коректно відображається на мобільних пристроях.

Для хостингу сайту був обраний github (github pages) - використовується пакет gh-pages, який пушить, обрану директорію (public) з прапором force в гілку master чи gh-pages при виклику npm скрипта в терміналі командою yarn deploy(npm deploy).

Сайт розташований за адресою [ssm1le.github.io/](https://ssm1le.github.io/)

### 3.2. Розробка веб-додатку на Vue.js

Vue - це прогресивний фреймворк для створення користувацьких інтерфейсів. На відміну від фреймворків-монолітів, Vue створений придатним для поступового впровадження. Його ядро в першу чергу вирішує завдання рівня

уявлення (view), що спрощує інтеграцію з іншими бібліотеками та існуючими проектами. З іншого боку, Vue повністю підходить і для створення складних односторінкових додатків (SPA, Single-Page Applications), якщо використовувати його спільно з сучасними інструментами та додатковими бібліотеками.

Для встановлення Vue в терміналі необхідно написати наступну команду:

```
npm install -g @vue/cli
```

Для створення нового сайту: *vue create*

Vue запускає середу розробки, доступну на localhost:8000 по команді в терміналі:

```
vue-cli-service serve
```

*vue-cli-service build* - Vue виконає оптимізовану збірку для сайту, генеруючи статичні HTML і зібрані JavaScript файли, іншими словами білд.

Також в файлі package.json написані такі скрипти:

“serve”: “vue-cli-service serve” – запуск локального сервера,

”build”: “vue-cli-service build” – складання продукту,

“lint”: “vue-cli-service lint” – форматування коду,

“deploy”: “deploy.sh” – скрипт для деплою проекту на хостинг.

Структура проекту зображена на рисунку 3.3.

Крім файлу package.json, в корені проекту створюється файл vue.config.js, в якому вказані налаштування для деплою версій додатку.



Рис. 3.3. Структура проекту Vue.js

Файл babel.config.js – конфіг для компіляції babel.

Файл `deploy.sh` – `bush` скрипт для деплою проекту на `github`(збирає білд, комітіть звідини у гудку `gh-pages` та пушить `ix`).

Файл `vue.config.js` – конфіг для білдера, локальний тестовий сервер чи сервер для деплою.

Основна директорія – `SRC`. У ній були створені такі директорії:

`Assets` – директорія з картинками

`Components` – директорія з компонентами

- `v-actions` – панель навігації в `header'e`;
- `v-caption` – заголовки до секцій;
- `v-card-event-item` – картка відповідної події на головній сторінці веб-додатку;
- `v-card-event` – картка відповідної події на сторінці події ресторану;
- `v-card-menu-item` – картка відповідної позиції меню ресторану на сторінці меню;
- `v-card-menu` – картка відповідної позиції меню ресторану на головній сторінці веб-додатку;
- `v-footer` – підвал сайту;
- `v-header` – шапка сайту;
- `v-carousel` – віджет з зображеннями на головній сторінці;

`Data` – директорія з даними (меню ресторану, події, свята та інші данні);

`Style` – директорія зі стилями додатку(конфіг стилів, шрифти та загальні стилі);

`Views` – обгортка для всіх сторінок, таких як:

- `Contacts` – сторінка контактів (контакти та компонент `vue2-google-maps`;
- `Events` – сторінка з всіма подіями закладу
- `Event-item` – опис окремої події
- `Menu` – сторінка з меню закладу
- `Menu-item` – опис окремої страви з меню

– Main – головна сторінка сайту ресторану

router.js – в цьому файлі прописані статичні та динамічні шляхи до сторінок (використовувався компонент vue-router)

Сайт написаний з використанням адаптивної верстки, тим самим коректно відображається на мобільних пристроях.

Для хостингу сайту був обраний github (github pages) - використовується пакет gh-pages, який пушить, обрану директорію (dist) з прапором force в гілку gh-pages при виклику npm скрипта в терміналі командою yarn run deploy(npm run deploy).

Сайт розташований за адресою [ssm1le.github.io/cuba-site-vue/](https://ssm1le.github.io/cuba-site-vue/)

### 3.3. Проведення вимірів

Крок 1. Налаштування DevTools (рисунок 3.4):

Відкрити сторінку. Поки сторінка знаходиться у фокусі, натиснути Command + Option + I (Mac) або Control + Shift + I (Windows, Linux), щоб відкрити DevTools на сторінці. У DevTools, клік на Network. включити скріншоти. DevTools робить скріншоти під час завантаження сторінки, щоб відобразити час.

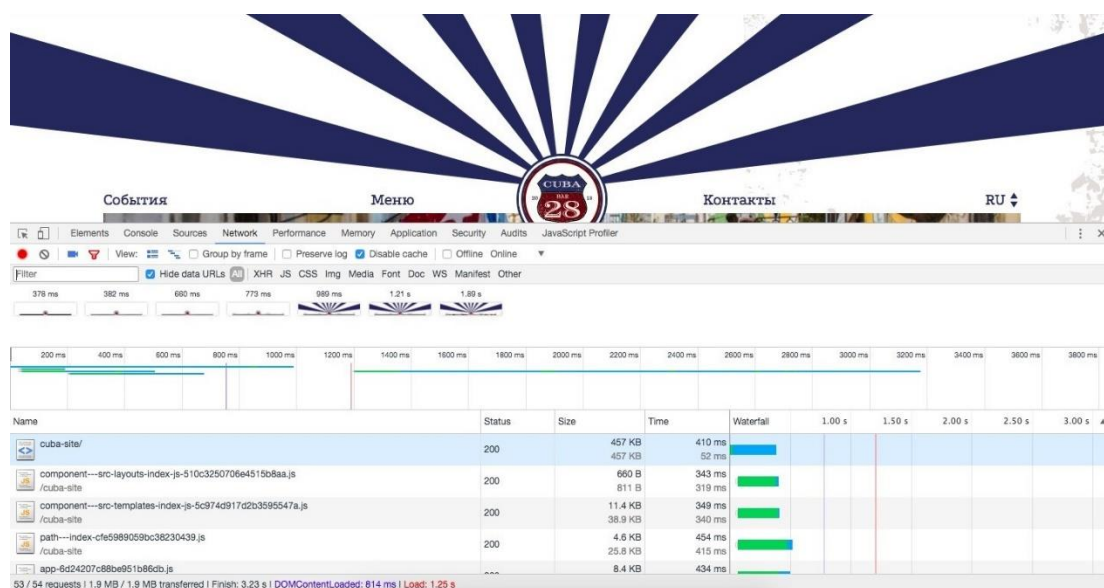


Рис. 3.4. Панель “Chrome DevTools”, відкрита поруч зі сторінкою, яку необхідно діагностувати.

## Крок 2. Емулювання роботи мобільних пристроїв.

Тестування продуктивності мережі на ноутбуці або на робочому комп'ютері може давати невірні результати. Таке інтернет-з'єднання набагато швидше, ніж у мобільного користувача, а браузер може кешувати ресурси з попередніх відвідувань.

Тому необхідно встановити прапорець Відключити кеш. Коли цей прапорець включений, DevTools не обслуговує ніяких ресурсів з кеша, а ще краще проводити тестування в режимі інкогніто. Це більш точно емулює поведінку того, як користувачі вперше переглядають сторінку.

У спадному меню, в якому за замовчуванням варто Online, необхідно вибрати Slow 3G. DevTools дроселює мережеве з'єднання, щоб імітувати звичайну швидкість 3G. Таким чином, мобільні користувачі використовують сайт в місцях з поганими підключеннями (рисунк 3.5).

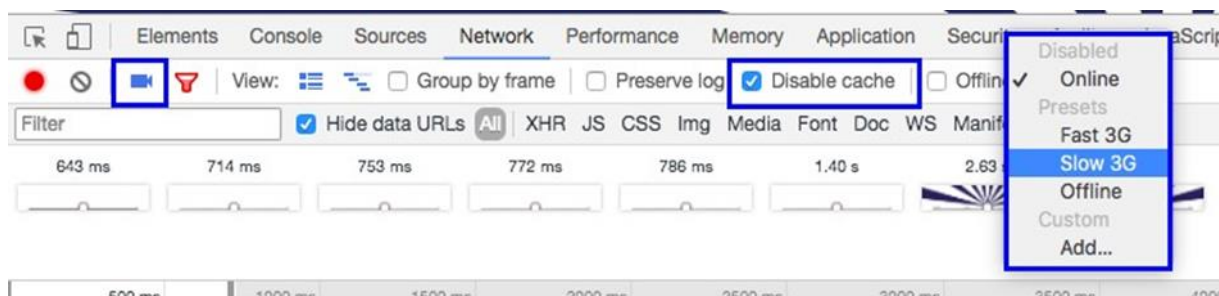


Рис. 3.5. Панель “Chrome DevTools”

Панель “Chrome DevTools”, яка налаштована для наслідування використання мобільними користувачами. Скріншоти, відключення кеша і дроселювання виділені синім кольором, зліва направо, відповідно.

Це найгірший варіант. Якщо сторінка буде швидко завантажуватися при встановлених налаштуваннях, вона буде швидкою для всіх користувачів.

## Крок 3. Аналіз запитів

Необхідно з'ясувати, що робить сторінку повільною, перезавантажуючи сторінку і аналізуючи вхідні запити. Знайти сценарії блокування рендеринга.

Коли браузер зустрічає тег `<script>`, він повинен призупинити рендеринг і виконати скрипт негайно. Тому необхідно знайти сценарії, які не потрібні для

завантаження сторінки і зробити їх асинхронними або відкласти їх виконання, щоб прискорити час завантаження.

### 3.4. Результати проведення тестів продуктивності та їх аналіз

Аналіз продуктивності проводився засобами DevTools браузера google chrome і сервісом gtmetrix (gtmetrix.com, з вибором геолокації London, UK і проведенням вимірів на звичайній швидкості, а також з троттлінг на швидкостях 2G і 3G).

Для проведення тестів за допомогою сервісу gtmetrix досить вказати адресу сторінки яку необхідно протестувати і вказати настройки тестування - регіон і швидкість і запустити тест.

З DevTools браузера Google Chrome можна побачити, що всі файли завантажувалися по протоколу HTTP / 2.

У браузері Chrome, якщо всередині тега <head> є тег <link rel preload>, то браузер не чекаючи повного завантаження документа, почне завантаження ресурсів з цих тегів.

На головній сторінці є 1 блокуючий тег скрипт, вбудований в сторінку - це функція, яка чекає, коли завантажиться вся сторінка, коли відбудеться подія loaded. Ця функція виконує цикл для проходження по масиву рядків, де кожен рядок - це шляхи до js файлів, (спочатку тега <Head> прописані ці шляхи в тегах <link rel preload> - це необхідно для того, щоб в сучасних браузерах завантаження файлів почалася до того, як повністю виконається завантаження документа), ці теги script з'являються в DOM'е після того, як виконається повністю завантаження сторінки (таким чином реалізована відкладена завантаження скриптів). Після завантаження HTML починає завантажуватися файл стилів, в якому перераховані всі шрифти і їх накреслення, необхідні для сторінки, після завантаження css файлу, починають завантажуватися файли шрифтів, а паралельно із завантаженням файлів стилів починають завантажуватися всі картинки, які є на сторінці, вага зображень досить великий (кількість + розмір), час завантаження зображень значно збільшує час завантаження сторінки.

Для зменшення часу завантаження можна вирішити наступні проблеми: На сторінці спочатку в тег style в head - додані посилання на шрифти - 2 шрифти - один для заголовків (вага шрифту bold), а другий для контенту (вага шрифту normal), для поліпшення продуктивності необхідно написати функцію, яка додасть на сторінку залишилися шрифтів. Таким чином, спочатку весь текстовий контент відобразиться шрифтом з вагою regular, а заголовки з вагою bold, а після повного завантаження всіх шрифтів, довантажиться italic, bold italic для текстового контенту.

В рамках даної роботи була реалізована технологія для збільшення продуктивності сайту завдяки відкладеної завантаження зображень.

У зв'язку з тим, що користувачі можуть завантажувати в сервіс contentful фотографії, виконані на професійну техніку, це може істотно позначитися на продуктивності сайту. Тому попередньо до завантаження сайту завантажуються зображення розміром 4 кб, і тільки після завантаження сайту в тезі <img> в атрибуті <srcset> замінюється посилання на зображення в реальному розмірі. Тим самим збільшується швидкість завантаження.

Також додано кешування сторінок. Нижче в таблиці 1 наведені результати тестів продуктивності на підключенні 16 мегабіт в секунду і з троттлінг (2G / 3G - gtmetrix, Slow 3G - Google Chrome).

Таблиця 3.1

Результати тестів продуктивності на підключенні 16 мегабіт в секунду і з троттлінг (2G / 3G - gtmetrix, Slow 3G - Google Chrome) до оптимізації.

	Інструменти для тестування	Головна сторінка		Події		Меню		Контакти		Одна подія	
		Швидкий зв'язок	троттлінг 2G/3G	Швидкий зв'язок	троттлінг 2G/3G	Швидкий зв'язок	троттлінг 2G/3G	Швидкий зв'язок	троттлінг 2G/3G	Швидкий зв'язок	троттлінг 2G/3G
Час на редирект	gtmetrix	0 мс	0 мс/0 мс	41 мс	3,1 с / 1,5 с	47 мс	3,1 с / 1,5 с	37 мс	3,1 с / 1,5 с	54 мс	3,1 с / 1,5 с
	google chrome	0 мс	63,51 мс	0,33 мс	0,52 мс	0,38 мс	0,52 мс	0,78 мс	0,5 с	0,84 мс	1,25 с



Продовження табл.3.1.

Тривалість з'єднання	gtmetri x	59 мс	1,8 с / 0,с	0 мс	1 мс / 1 мс	1 мс	1 мс / 0 мс	0 мс	0 мс / 1 мс	0 мс	1 мс / 1 мс
	google chrome	0,2 мс	105,99 мс	0,17 мс	0,25 мс	0,17 мс	0,25 мс	0,27 мс	0,2 с	0,18 мс	0,54 с
Час роботи сервера	gtmetri x	5 мс	0,5 с / 223 мс	3 мс	0,5 с / 219 мс	2 мс	0,5 с / 219 мс	1 мс	0,5 с / 218 мс	3 мс	0,5 с / 219 мс
	google chrome	-	-	-	-	-	-	-	-	-	-
Час до першого байту	gtmetri x	64 мс	2,3 с / 1,1 с	44 мс	3,6 с / 1,7 с	50 мс	3,6 с / 1,7 с	38 мс	3,6 с / 1,7 с	57 мс	3,6 с / 1,7 с
	google chrome	55,1 мс	2,03 с	138,84 мс	2,03 с	138,86 мс	2,03 с	151,65 мс	2,02 с	144,14 мс	2,02 с
Час першого відтворення	gtmetri x	0.6 с	7,7 с / 2,4 с	0,7 с	9 с / 3 с	0,5 с	8,8 с / 4 с	276 мс	8,8 с / 3,1 с	0,6 с	8,6 с / 3 с
	google chrome	1,06 мс		229 мс	3,55 с	210 мс	3,55 с	187 мс	4,7 с	181 мс	5,1 с
Завантаження DOM	gtmetri x	249 мс	9,5 с / 2,7 с	203 мс	10,9 с / 3,8 с	165 мс	10,9 с / 3 с	117 мс	10,7 с / 3,1 с	148 мс	10,9 с / 3 с
	google chrome	266 мс	14,59 с	294 мс	15,13 с	249 мс	15,13 с	783 мс	9,19 с	325 мс	7,46 с
Час до першого інтерактиву	gtmetri x	249 мс	9,5 с / 2,7 с	203 мс	10,9 с / 3,8 с	165 мс	10,9 с / 3 с	117 мс	10,7 с / 3,1 с	148 мс	10,9 с / 3 с
	google chrome	266 мс	14,59 с	294 мс	15,13 с	249 мс	15,13 с	783 мс	9,19 с	325 мс	7,46 с

Продовження табл.3.1.

Час повного завантаження	gtmetri x	1.3 с	65,5 с / 10,8 с	1,1 с	47 с / 8,8 с	0,9 с	37,9 с / 7,5 с	0,8 с	56,8 с / 11,1 с	0,7 с	23,9 с / 5,1 с
	google chrome	1,57 с	18,3 с	1,15 с	17,9 с	1,02 с	17,9 с	1,51 с	13,51 с	810 мс	11,71 с

На рисунку 3.6. наведено відображення отриманого часу на завантаження головної сторінки сайту на повільній швидкості

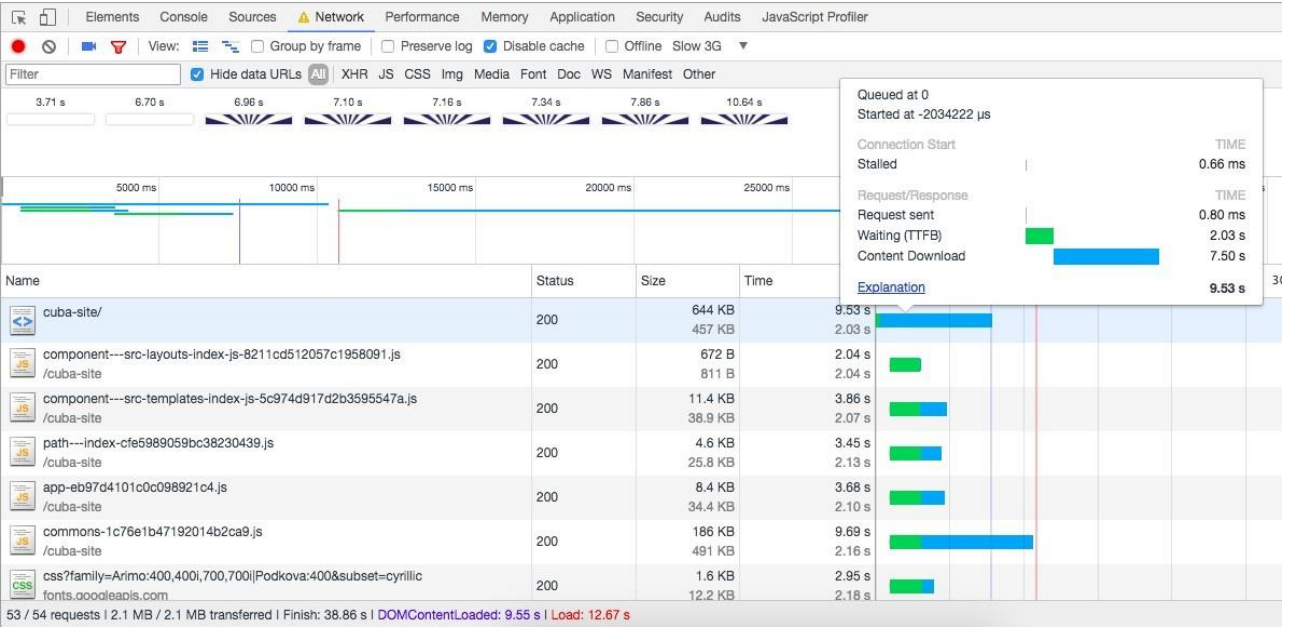


Рис. 3.6. Панель Chrome DevTools Network завантаження сторінки на повільній швидкості.

Виходячи з проведеного гвиміру, можна побачити, що час DOMContentLoaded 9,55 секунд, а повне завантаження 12,67 секунд, час до першого байта 2,03 секунди, перший інтерактив стався на 3,71 секундi.

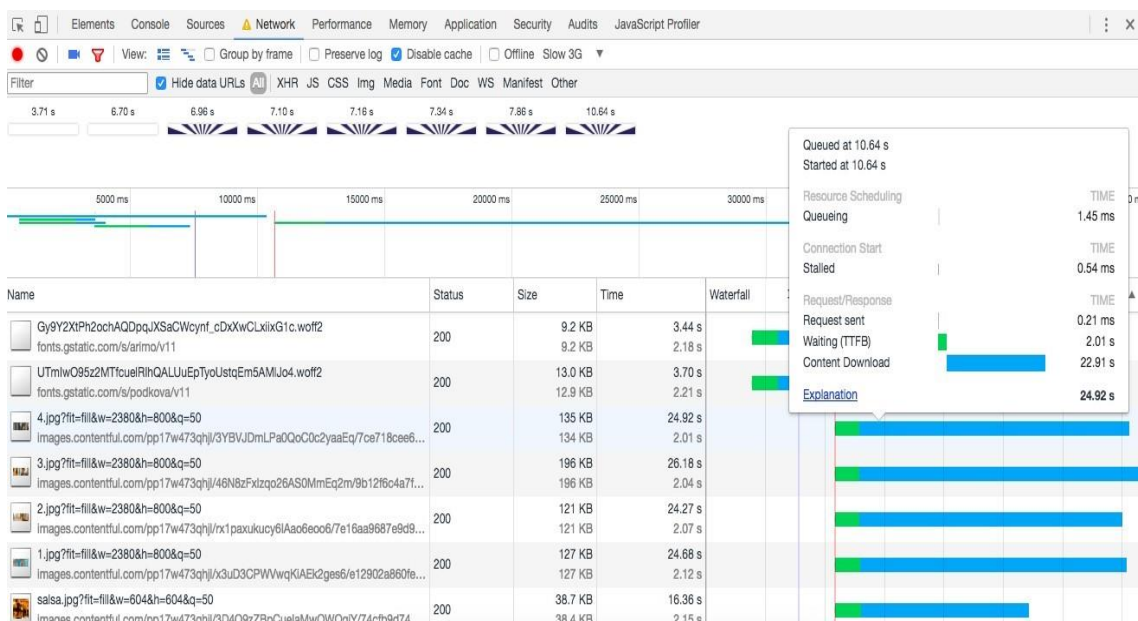


Рис. 3.7. Панель Chrome DevTools Network завантаження зображень в повному розмірі на повільній швидкості.

На рисунку 3.7. відображений час початку дозавантаження зображень - через 10,64 секунди, які важили спочатку 4 кілобайти. Ми також можемо помітити, що повне завантаження сторінки сталася через 38,86 секунд, що складає 67%, таким чином нам вдалося скоротити час очікування користувачем на 26,19 секунд, тобто користувач уже міг весь цей час взаємодіяти зі сторінкою. На рисунку 3.8 наведено результати завантаження головної сторінки на швидкості завантаження близько 16 мегабіт в секунду.

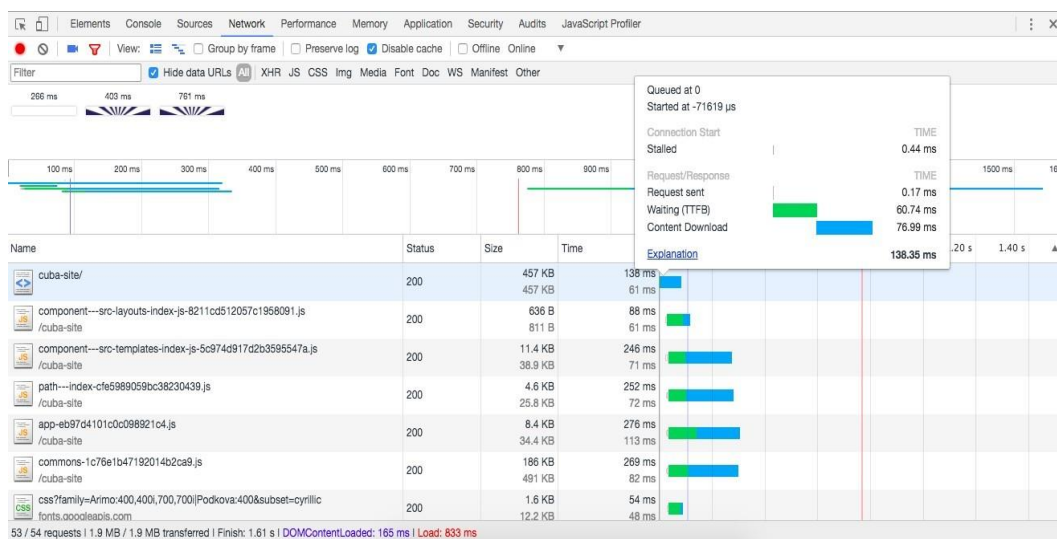


Рис. 3.8. Панель Chrome DevTools Network завантаження сторінки на швидкості ~ 16 мб / с.

За результатами проведеного виміру, можна побачити, що час DOMContentLoaded 165 мілісекунд, а повне завантаження 833 мілісекунди, час до першого байта 60,74 мілісекунди, перший інтерактив став на 266 мілісекунді. На рисунку 3.9. відображений час початку дозавантаження зображень - через 774 мілісекунди, які важили спочатку 4 кілобайти.

Повне завантаження сторінки з важкими зображеннями сталася через 1,61 секунди, тобто нам вдалося заощадити 777 мілісекунди, а це майже 50%.

Це дуже гарний результат - 50 відсотків до завантаження сторінки сайту.

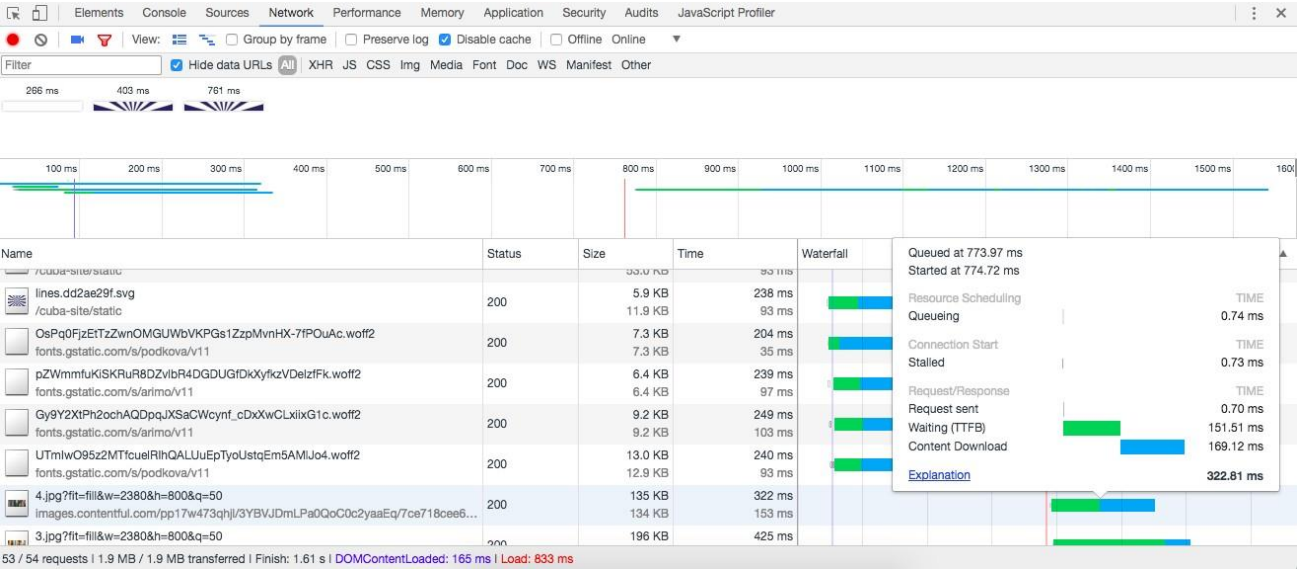


Рис. 3.9. Панель Chrome DevTools Network завантаження зображень в повному розмірі на швидкості ~ 16 мб / с.

Таблиця 3.2.

Результати тестів продуктивності на підключенні 16 мегабіт в секунду

	Інструменти для тестування	Головна сторінка		Події		Меню		Контакти		Одна подія	
		Швидкий зв'язок	троттлінг 2G/3G	Швидкий зв'язок	троттлінг 2G/3G	Швидкий зв'язок	троттлінг 2G/3G	Швидкий зв'язок	троттлінг 2G/3G	Швидкий зв'язок	троттлінг 2G/3G
Час на редирект	gtmetrix	0 мс	0 мс / 0 мс	64 мс	3,1 с / 1,5 с	46 мс	3,1 с / 1,5с	39 мс	3,1 с / 1,5 с	41 мс	3,1 с / 1,5с
	google chrome	0 мс	0 мс	0,34 мс	0,48 мс	0,47 мс	0,25 мс	0,82 мс	0,44 мс	0,33 мс	0,24 мс

Продовження таблиці 3.2

Тривалість з'єднання	gtmetri x	54 мс	1,8 с / 0,9 с	1 мс	1 мс / 1 мс	1 мс	1 мс / 1 мс	1 мс	1 мс / 1 мс	1 мс	1 мс / 1 мс
	google chrome	0,17 мс	0,80 с	1,52 мс	0,43 мс	0,62 мс	0,91 мс	0,43 мс	0,19 мс	0,17 мс	82 мс
Час роботи сервера	gtmetri x	11 мс	0,5 с / 226 мс	2 мс	0,5 с / 218 мс	3 мс	0,5 с / 220 мс	3 мс	0,5 с / 219 мс	2 мс	0,5 с / 221 мс
	google chrome	-	-	-	-	-	-	-	-	-	-
Час до першого байту	gtmetri x	65 мс	2,3 с / 1,1 с	67 мс	3,6 с / 1,8 с	50 мс	3,6 с / 1,7 с	43 мс	3,6 с / 1,7 с	44 мс	3,6 с / 1,7 с
	google chrome	60,74 мс	2,03 с	148,6 9 мс	2,02 с	56,68 мс	2,01 с	144 мс	2,02 с	143,1 3 мс	2,47 с
Час першого відтворення	gtmetr i x	266 мс	7,7 с / 2,7 с	0,7 с	8,8 с / 3,5 с	0,6 с	8,7 с / 3,2 с	303 мс	8,8 с / 3,0 с	274 мс	8,6 с / 3,0 с
	google chrome	291 мс	3,71 с	184 мс	3,56 с	186 мс	3,49 с	204 мс	3,11 с	490 мс	6,47 с
Завантаження DOM	gtmetr i x	182 мс	9,9 с / 2,4 с	173 мс	10,8 с / 3 с	316 мс	11 с / 3,1 с	154 мс	11 с / 3,0 с	152 мс	10,8 с / 3,0 с
	google chrome	165 мс	9,55 с	123 мс	9,04 с	122 мс	7,61 с	303 мс	8,8 с	215 мс	10,81 с
Час до першого інтерактиву	gtmetr i x	182 мс	9,9 с / 2,4 с	173 мс	10,8 с / 3 с	316 мс	11 с / 3,1 с	154 мс	11 с / 3,0 с	152 мс	10,8 с / 3,0 с
	google chrome	165 мс	9,55 с	123 мс	9,04 с	122 мс	7,61 с	303 мс	8,8 с	215 мс	10,81 с



Продовження таблиці 3.3

Час до першого байту	gtmetrix	55 мс	2,3 с / 1,1 с	61 мс	3,6 с / 1,8 с	51 мс	3,5 с / 1,8 с	40 мс	3,1 с / 1,6 с	44 мс	3,6 с / 1,7 с
	google chrome	50,12 мс	2,03 с	65,69 мс	2,02 с	56,68 мс	2,01 с	144 мс	2,02 с	143,13 мс	2,47 с
Час першого відтворення	gtmetrix	270 мс	7,7 с / 2,7 с	311 мс	8,8 с / 3,5 с	0,6 с	8,7 с / 3,2 с	303 мс	7,8 с / 3,0 с	254 мс	8,3 с / 3,1 с
	google chrome	280 мс	3,71 с	286 мс	3,56 с	186 мс	3,49 с	204 мс	3,11 с	495 мс	6,57 с
Завантаження DOM	gtmetrix	350 мс	9,9 с / 2,4 с	485 мс	10,8 с / 3 с	316 мс	11 с / 3,1 с	402 мс	11 с / 3,0 с	152 мс	10,8 с / 3,0 с
	google chrome	390 мс	9,55 с	385 мс	9,04 с	122 мс	7,61 с	402 мс	8,8 с	215 мс	10,81 с
Час до першого інтерактиву	gtmetrix	430 мс	9,9 с / 2,4 с	512 мс	10,8 с / 3 с	416 мс	11 с / 3,1 с	454 мс	11 с / 3,0 с	522 мс	10,8 с / 3,0 с
	google chrome	414 мс	9,55 с	403 мс	9,04 с	422 мс	7,61 с	463 мс	7,8 с	515 мс	6,81 с
Час повного завантаження	gtmetrix	850 мс	65,3 с / 15,27 с	834 мс	49,1 с / 12,58 с	843 мс	50,7 с / 13,3 с	1,378 с	54,9 с / 13,9 с	777 мс	42,2 с / 12,2 с
	google chrome	890 мс	14,39 с	830 мс	13,43 с	855 мс	12,34 с	1,32 с	14,97 с	796 мс	12,07 с

### Висновки до розділу 3

В третьому розділі була проведена розробка веб-додатків на javascript фреймворках, таких як Gatsby (під капотом все то же React від facebook) та Vue

(від розробника Angular.js у минулому). Ці фреймворки є компонентними, реактивні, перший – JSX (препроцесор, синтаксис от xml), та другий – звичайний HTML.

Були проведені досліді на продуктивності на підключенні 16 мегабіт в секунду і з троттлінг (2G / 3G - gtmetrix, Slow 3G - Google Chrome) до та після оптимізації.

Грунтуючись на проведених дослідженнях, можна сказати, що методики для збільшення продуктивності веб-додатків дійсно дали результат! Я вважаю що їх необхідно дотримуватися при розробці веб-додатків.



## ВИСНОВКИ

В результаті даної роботи були виконані наступні завдання:

Розроблено веб-додатки для ресторану з використанням фреймворку React і сервісу для зберігання даних `contnetful` та фреймворку `Vue.js` для порівняння. Веб-додатки з адаптивною версткою.

Також були розроблені методи зниження ресурсоемності додатків, за рахунок застосування компонентної архітектури.

В рамках магістерської атестаційної роботи був проведений порівняльний аналіз продуктивності, який проводився за швидкістю завантаження сайту за наступними показниками:

- час на редирект;
- тривалість з'єднання;
- час роботи сервера;
- час до першого байта (TTFB);
- час першої відтворення;
- завантаження DOM;
- час до першого інтерактиву;
- час повного завантаження.

При реалізації веб-додатків та проведення тестування продуктивності була розроблена технологія по збільшенню продуктивності та зменшення швидкості завантаження сторінок за рахунок відкладеної завантаження зображень, шляхом спочатку завантаження зображень розміром 4 кілобайти, а після завантаження DOM відбувається до завантаження зображень в повному розмірі.

В результаті проведеного дослідження були зроблені наступні висновки, що для збільшення продуктивності веб-додатків необхідно застосування таких патернів:

- оптимізація коду;
- оптимізація підключення чергово стилів та скриптів;
- відкладена завантаження JavaScript (`defer`) та поділ скриптів;

- кешування сторінок;
- оптимізація завантажуються зображень під різні пристрої та стиснення картинок через спеціальні ресурси та сервіси.

В порівнянні Vue.js та Gatsby.js – однозначно сказати, що якийсь із цих фреймворків краще - неможливо, так як кожен для себе знайде якісь плюси.

## ЛІТЕРАТУРНІ ДЖЕРЕЛА

### Книги

1. Марка Д., Методология структурного анализа и проектирования / Д. Марка, К. МакГоуэн. – М.: МетаТехнология, 1993. – 240 с.
2. Синицын С. Верификация программного обеспечения. / Синицын С. В., Налютин Н. Ю.— М.: БИНОМ, 2008. — 368 с.
3. Гленфорд Майерс. Искусство тестирования программ / Гленфорд Майерс, Том Баджетт, Кори Сандлер. –[3-е изд.] – Москва: Диалектика, 2012. – 345 с. – (3).
4. Канер Кем. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений. / Канер Кем, Фолк Джек, Нгуен Енг Кек. — Киев: ДияСофт, 2001. — 544 с.

### Електронні ресурси

1. Довідниково-інформаційний портал htmlbook – Режим доступу до ресурсу: <http://htmlbook.ru/>.
2. Веб-сервіс для спільної розробки програмного забезпечення – GitHub.
3. Сучасний підручник Javascript – Режим доступу до ресурсу: <https://learn.javascript.ru/>.
4. Стаття про garbage collector: хттпс – Режим доступу до ресурсу: <https://habrahabr.ru/post/309318/>.
5. Analysis and Evaluation of Web Application Performance Enhancement Techniques – Режим доступу до ресурсу: [https://link.springer.com/chapter/10.1007/978-3-319-08245-5\\_3](https://link.springer.com/chapter/10.1007/978-3-319-08245-5_3).
6. Contentful – Режим доступу до ресурсу: <http://www.contentful.com/>.
7. Client Side Performance Testing – Режим доступу до ресурсу: <http://blog.dataart.com/client-side-performance-testing/>.
8. Fundamentals of Web Application Performance Testing – Режим доступу до ресурсу: <https://msdn.microsoft.com/en-us/library/bb924356.aspx>.

9. Fundamentals of Web Application Performance Testing – Режим доступа до ресурсу: <https://stackify.com/fundamentals-web-application-performance-testing/>.
10. Gatsby Documentation – Режим доступа до ресурсу: <https://www.gatsbyjs.org/>.
11. GraphQL Documentation – Режим доступа до ресурсу: <http://graphql.org/learn/>.
12. Web performance testing: Top 12 free and open source tools to consider – Режим доступа до ресурсу– Режим доступа до ресурсу: <https://techbeacon.com/web-performance-testing-top-12-free-open-source-tools-consider/>.
13. Gtmetrix – Режим доступа до ресурсу: <https://gtmetrix.com>.
14. HTTP archive – Режим доступа до ресурсу: <http://httparchive.org/>.
15. Image Optimization – Режим доступа до ресурсу: <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization>.
16. Linter – Режим доступа до ресурсу: <https://eslint.org/>.
17. Mindmap – Режим доступа до ресурсу: <https://app.mindmup.com>.
18. Optimizing Content Efficiency – Режим доступа до ресурсу: <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/>.
19. Performance Get Started With Analyzing Runtime Performance – Режим доступа до ресурсу: <https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/>.
20. React Documentation – Режим доступа до ресурсу: <https://facebook.github.io/react/>.
21. Understanding Low Bandwidth and High Latency – Режим доступа до ресурсу: <https://developers.google.com/web/fundamentals/performance/poor-connectivity/>.